# C++ ו- OOP

## למתכנת המקצועי

הוראות התקנה והפעלה של הדיסקט תמצא בנספח.
עיין בהן בתשומת לב לפני ההתקנה.
במהלך הספר תמצא התייחסות לקבצים.

עורך ראשי: **יצחק עמיהוד**
עריכה מקצועית: **צור ריכטר-לוין**
עריכה לשונית ועיצוב: **שרה עמיהוד**
עיצוב עטיפה: **סטודיו מצגר**



**לשם שטף הקריאה כתוב ספר זה בלשון זכר בלבד. ספר זה מיועד לגברים ונשים כאחד ואין בכוונתנו להפלות או לפגוע בציבור המשתמשים/ות.**

- **טלפון: 09-9564716 24 שעות**
- **פקס: 09-9571582 24 שעות**
- **דואר אלקטרוני: info@hod-ami.co.il**
- **אתר באינטרנט: www.hod-ami.co.il**

# C++ ו- OOP
# למתכנת המקצועי

**שמעון כהן**




**הוצאת הוד-עמי**
**לספרי מחשבים**

# C++ and OOP for the Advanced Programmer

By **Shimon Cohen**

Editor: **I. Amihud**




**הוצאת הוד-עמי**
**לספרי מחשבים בע"מ**
ת.ד. 6108 הרצליה 46160
טלפון: 09-9564716 פקס: 09-9571582





הודפס בישראל
תשנ"ז, 1997




**HOD-AMI Ltd.**
P.O.B. 6108, Herzliya
ISRAEL, 1997

ISBN 965-361-142-9 מסת"ב

# תוכן עניינים מקוצר

# תוכן העניינים

## מבוא



## פרק 1 סקירה של שפת C

## פרק 2 תכנות מוכוון אובייקטים



## פרק 3 המחלקה

## פרק 4 קלט ופלט



## פרק 5 ירושה ככלי לשימוש חוזר בקוד

## פרק 6 פולימורפיזם

## פרק 7 תבניות ב-++C



## פרק 8 טיפול בשגיאות ועוד

## פרק 9 חידושים ב-++C

# פרק 10 ספריית תבניות סטנדרטיות STL



# פרק 11 מבני נתונים סדרתיים ב-STL

# פרק 12 מבנים שאינם סדרתיים ב-STL



# פרק 13 שילוב סוגי תכנות

# נספח



# אינדקס

# מבוא

## על ++C

כשכותבים ספר, נותנים את הדעת למי הוא מיועד, כי קהל היעד קובע את הרמה והסגנון של הספר. **הספר שלפניכם מיועד למתכנתים בשפת ++C שרוצים להגיע לרמת מיומנות גבוהה ולהתקדם להיות מתכנתים מקצועיים באמת. הוא גם מיועד למתכנתי C ושפת פסקל, שאינם מכירים את שפת התכנות ++C**. לטובת מתכנתי C ניתן פרק הגירה המסכם את אפשרויות שפת C, עם הצבעה על המקבילות להן בשפת ++C. במהלך הלימוד יש הסברים והתייחסויות הקשורים לשפת C, מכיון ששפת ++C התפתחה כהמשך לשפת זו. מתכנתים ב**שפת פסקל** יוכלו גם כן להשתמש בספר, אך ייתכן שקצב הלימוד שלהם יהיה איטי יותר בתחילה. לאלה האחרונים, ההמלצה היא לקרוא בעיון את הפרק העוסק בשפת C.

**שפת התכנות ++C והמושג OOP - תכנות מוכוון אובייקטים - חד הם.** אפשר לומר ששפת ++C מבטאת את המימוש של עקרונות OOP, ולכן לא ניתן להפריד בין השניים. **התכונות והכלים של ++C נגזרים מעקרונות OOD (עיצוב מוכוון אובייקטים) ו-OOP** ולכן, אי אפשר ללמוד ++C ללא OOP, ואי אפשר ללמוד OOP ללא ++C. במהלך הלימוד והרחבת הידע נעסוק בשני תחומים: **תחום התכנות בשפת ++C ותחום תפישת התכנות המתקדמת - OOP**, אשר מהווה כאמור, את התשתית לשפת התכנות.

קיימים ספרים לא מעטים על ++C, רובם מתארים את השפה ברמה בסיסית ואחרים מתארים אותה ברמה מתקדמת יותר. ספר זה מתחיל מרמה בסיסית של ++C, אך מניח שלקורא יש רקע בסיסי כלשהו בתכנות. על כן, הוא ממשיך מיד לנושאים מתקדמים, למושגים חדשים ולחידושים בשפה שטרם מצאו ביטויים ברבים מהספרים המתקדמים. מטרת הספר היא להפוך **מתכנתי ++C "רגילים" למתכנתים מקצועיים** בשפה זו, השולטים בטכניקות תכנון ותכנות חדשות ומתקדמות.

++C היא שפה מתפתחת. התקן שלה אינו גמור עדיין (לפחות בעת כתיבת שורות אלו), אך קרוב לסיומו. שיטות וטכניקות חדשות עדיין מתוספות ולכן, ספרים חדשים המלמדים טכניקות ומונחים חדשים הם כורח המציאות.

שפות התכנות C ופסקל הן שפות פשוטות ובעלות יכולות מוגבלות. מתכנת ממוצע יכול להשתלט ולהתמחות בשפות אלו בזמן קצר יחסית, במספר חודשים מועט. עם ++C, לעומת זאת, המצב שונה. גם מתכנתים המשתמשים בשפה זו מספר שנים, אינם מכירים את כל אפשרויותיה. הזמן המשוער למתכנת ממוצע להשתלט על יכולות השפה מוערך בשנים. מתכנת שמכיר את C יכול לכתוב תוכניות ב-++C כמעט באופן מיידי, אך כדי לכתוב תוכניות יעילות ובאיכות גבוהה, דרוש זמן לימוד רב. **ספר זה מלמד את השיטות והטכניקות הדרושות למתכנת כדי שיוכל לנצל את השפה ביעילות.** לימוד עקבי יכול לקצר את זמן הלימוד באופן משמעותי.

++C היא שפה הייברידית, כלומר, שפה שממזגת מספר גישות. היא יורשת מ-C את כל המבנים והתכונות שלה, אך בנוסף לאלה היא מציגה יכולות חדשות, כגון **תכנות מוכוון אובייקטים** (Object Oriented Programming) **ותכנות גנרי** (Generic Programming). למרות ששני המונחים האחרונים נשמעים זהים, הם שונים. הסבר על כך תמצא בפרקים המתאימים.

שפת ++C היא שפה מוכלאת (הייבריד) ולכן היא מאפשרת לכתוב תוכניות דמויות C בשפה ++C. רבים חושבים שאפשרות זו היא חיסרון. קיימות שפות מוכוונות אובייקטים אחרות, שאינן מאפשרות כתיבה אחרת, פרט לכתיבה בשיטה זו. חסידי שפות אלו טוענים ששפת ++C מאפשרת חופשיות רבה מידי למתכנת, ואף מאפשרת להשתמש במבנים שאינם טובים מבחינת הנדסת תוכנה. מבנים אלה הן **פונקציות** שאינן קשורות לאובייקטים.

בשפת אובייקטים טהורה קיימים רק אובייקטים, כל הפונקציות נקראות **שיטות (methods), ואינן** עומדות בפני עצמן, אלא **שייכות לאובייקט כלשהו**. אובייקט מייצג מבנה נתונים כלשהו ופעולות המבוצעות עליו, ואלה קשורים בקשר בלתי ניתן להפרדה. תכונה זו היא יתרון במקרים רבים, אבל ייתכנו גם מקרים בהם נרצה לכתוב פונקציות **שאינן** קשורות לאובייקט. הדבר יעיל במיוחד כשרוצים לכתוב פונקציות גנריות (כלליות), הפועלות על מגוון רחב של אובייקטים, ללא תלות בסוג. היכולות לבצע דברים כאלה אינה נתונה בשפות אחרות. יכולת זו הולידה את המונח **תכנות גנרי**, שנתמך ב-++C, אבל לא בשפות אובייקטים אחרות.

**תכנות גנרי** (Generic Programming) מאפשר אם כן לכתוב פונקציות, שהפרמטרים שלהן וסוג האובייקטים שהן מקבלות משתנה ומותאם לפי סוג השימוש שנעשה בהם. תכונה זו מאפשרת לכתוב פונקציות גנריות הפועלות על סוגים רבים של אובייקטים שאינם קשורים זה לזה, כמו למשל, פונקציה שמעתיקה תחום אחד למשנהו. ה"תחום" יכול להיות מערך, רשימה, עץ חיפוש וכדומה. פעולת ההעתקה אינה תלויה בסוגים אלה, כל עוד הם מספקים ממשק קבוע של פעולות. הדבר חוסך מהמתכנת שכפול של פונקציות מספר רב של פעמים, אחת לכל סוג של מבנה נתונים, או לכל סוג אובייקט.

בשפות אובייקטים טהורות אחרות כמו **SmallTalk**, כל סוגי הנתונים הם מחלקות היורשות מאובייקט גנרי כלשהו. לא כך הדבר ב-++C, שבה יש את כל סוגי הנתונים שקיימים ב-C, וגם סוגי נתונים חדשים, או הרחבות של השפה. סוגי נתונים אלה הן **מחלקות** (Classes) אשר נקראות גם: **סוגי נתונים המוגדרים על ידי המשתמש** (User Defined Types).

מתכנני ++C הקדישו מחשבה רבה ליעילות השפה. ++C אינה נופלת משפת התכנות C, שנחשבת לשפה היעילה ביותר בין שפות התכנות המקובלות, ויש מקרים שאפשר להגיע בה לרמת יעילות גבוהה יותר. עובדה זו נראית מפתיעה, ומרבית הקוראים יתקשו לקבל עובדה זו. **ספר זה מדגים את המקרים ואת השיטות שמאפשרים להגיע ליעילות כזו**, המושגת הודות ליכולת לכתוב פונקציות גנריות.

כדי להגיע ליעילות המתקרבת לשפת התכנות C חייבים להכיר את ++C היטב ולשלוט ברזי השפה (רבים לא מנצלים אותה בצורה יעילה). בין שפות האובייקטים, ++C היא היעילה ביותר. היא מאפשרת גישה לסיביות בודדות במילה, לפעול על מצביעים, וגם מאפשרת גישה לאוגרים של המחשב.

העיקרון העומד מאחורי ++C הוא שאי-שימוש ביכולות מסוימות של השפה לא יפגע במתכנת, או בתוכנית שהוא מפתח. מתכנתים יכולים להשתמש רק בחלקי השפה שהם מכירים, ללא כל צורך להכיר את כולה על בוריה.

# על הספר ומה שיש בו

הנושאים הנלמדים בספר זה רבים, ומכסים את כל היבטי השפה. הפרקים מחולקים ועולים ברמת הקושי, ככל שמתקדמים בקריאת הספר. ההנחה הבסיסית היא שקורא שמתקדם מפרק לפרק רוכש יותר ידע, ומאפשר לעלות ברמת הנושאים שהוא לומד.

**הפרק הראשון** מלמד על **המבנים ומשפטי הבקרה** ש- ++C יורשת מ-C. פרק זה מביא את הקורא לרמה הידע הדרושה בשפת C כדי שיוכל ללמוד את הנושאים האחרים שבספר.

**הפרק השני** מציג את המושג **תכנות מוכוון אובייקטים - OOP**. הפרק מתחיל בהצגת המונח **מודול**, שקדם לתכנות מוכוון אובייקטים. הפרק מציג את הסיבות והמוטיבציה שהולידו את הטכניקות של תכנות מוכוון אובייקטים.

**הפרק השלישי** מציג את המונח **מחלקה**. מונח זה התפתח מהמונח **מודול**. מחלקה ב-++C היא סוג נוסף שמגדיר המתכנת ובכך מרחיב את השפה. כלומר, ++C מאפשרת להגדיר אופרטורים למחלקות, או סוגים חדשים המוגדרים על ידי המתכנת, כך שסוגים אלה ייראו כמו סוגים בסיסיים של השפה. זהו מונח מרכזי בשפה זו ומהחשובים בשפות אובייקטים.

**הפרק הרביעי** מציג את **ספריית הקלט/פלט** של ++C. שפת התכנות עצמה אינה כוללת מבנים המאפשרים טיפול בקבצים. במקום זאת, מסופקת ספרייה תקנית

המאפשרת את הטיפול בקלט/פלט. שימוש בספרייה זו מאפשר לכתוב תוכניות ניידות שיכולות לרוץ על סוגים שונים של מחשבים ומהדרים, ללא צורך בשינוי.

**הפרק החמישי** מלמד את המושג **ירושה** (Inheritance). הפרק מציג את המונחים **ירושה, ירושה מרובה וירושה וירטואלית**. הפרק מציג נושאים רבים הקשורים לירושה, כגון סדר בנייה של אובייקטים, סדר פירוק של אובייקטים, מתי להשתמש בירושה וכיצד להשתמש בירושה בצורה טובה.

**הפרק השישי** מלמד את הנושא **פונקציות ווירטואליות**. פונקציות וירטואליות קשורות למונח פולימורפיזם, כלומר ריבוי-צורתיות. מונח זה, יחד עם מחלקות וירושה, מהווה בעיני רבים את הבסיס למונח תכנות מוכוון אובייקטים. פרק זה מציג גם את מודל האובייקטים וכיצד מושגת קשירה מאוחרת של פונקציות לאובייקטים.

**הפרק השביעי** מתאר את המונח **תבניות** (Templates) ב-++C. מונח זה הינו מהחשובים בשפה, אשר מאפשר **תכנות גנרי**, אחת מטכניקות התכנות החדשות שהופיעה לאחרונה ב-++C. פרק זה מסביר את עקרונות התכנות הגנרי.

**הפרק השמיני** מדגים את המונח **מצבים חריגים** (Exception Modes). מצבים חריגים הם תוספת חדשה יחסית ל- ++C. מתכנתים מעטים יודעים לטפל במצבים חריגים, בדרך נאותה. נושא זה אינו מוסבר בדרך כלל בספרי לימוד אחרים של השפה. בדרך כלל, מוסברות רק המכניקה ומילות המפתח של השפה. בספר זה, לעומת זאת, בנוסף למכניקה ומילות המפתח, מוסברות טכניקות תכנות וטיפול במצבים חריגים. גם מוצגות מספר מחלקות העוזרות בטיפול במצבים חריגים.

**הפרק התשיעי** מציג **נושאים חדשים** שהתוספו לשפה ושאינם נתמכים על ידי רוב המהדרים. בין הנושאים כלולים **מרחבי שמות**, **המרות של מצביעים**, **זיהוי סוגי אובייקטים בזמן ריצה** ועוד.

**הפרק העשירי** מהווה מבוא לספריית התבניות הסטנדרטית של ++C, **STL**. ספרייה זו הביאה לעולם את המונח **תכנות גנרי** ויצרה מהפכה באופן החשיבה והשימוש ב-++C. STL היא ספרייה יעילה במיוחד, יחסית לספריות האחרות הקיימות ב-++C וכדי להשתמש בה ביעילות דרוש ידע נרחב בתבניות. הספרייה יעילה בפעולותיה פי שתיים או שלוש מספריות אחרות. הספרייה תומכת ב**מכולות**, שהם אובייקטים המכילים אובייקטים רבים אחרים (כגון, רשימות). הספרייה תומכת באיטרטורים שמאפשרים לפעול על מכולות בצורות שונות, ומספקת מגוון רחב של אלגוריתמים גנרים. אלגוריתמים כמו sort, שממיין תחום נתונים, או אלגוריתם כגון unique, שמבטל מופעים כפולים של אובייקטים בתחום נתון. אלגוריתמים אלה פועלים על כל סוג נתונים כל עוד סוג הנתונים מקיים ממשק מסוים. כלומר, האלגוריתמים המסופקים בספרייה יכולים לתמוך במכולות של ספריות אחרות!

**הפרק האחד עשר** מציג את הנושא **מבנים סדרתיים** שמספקת STL. מבנים אלה כוללים **וקטור** (מערך חד-ממדי), **רשימה** ו**תור כפול**. כמו כן, מציג הפרק את האיטרטורים של מחלקות אלו. הפרק נותן הסבר על המבנה הפנימי של המחלקות וכללים מתי ואיך להשתמש בהן ביעילות.

**הפרק השניים עשר** דן ב**מבנים שאינם סידרתיים** ושמספקים חיפוש יעיל ומהיר של אובייקטים בתוכם. מבנים אלה מבוססים על עצי חיפוש בינארים מאוזנים. המחלקות שמתוארות בפרק זה הן: **set**, **multiset**, **map** ו-**multimap**. כמו כן, מתוארים האיטרטורים שלהם ומוצגות דוגמאות לשימוש במחלקות אלו.

בשני הפרקים האחרונים מודגמת, ומוצגת, שיטת התכנות הגנרית שהביאה STL לעולם. השיטה מוצגת לעומק ועקרונותיה מוסברים בפירוט.

**בפרק השלושה עשר** מתוארים מתאמים (adaptors) ושילוב בין תכנות גנרי לבין תכנות מוכוון אובייקטים. פרק זה מציג דוגמה המשלבת בין עקרונות התכנות מוכוון האובייקטים והתכנות הגנרי, ניהול מטלות ועוד.

רוב המתכנתים הלומדים את ++C, הופכים לחסידים נלהבים שלה. צריך להזכיר לקורא, **שהשפה היא כלי להשגת המטרה ולא המטרה עצמה**. המטרה היא לכתוב תוכניות שמקיימות דרישות מסוימות בצורה יעילה ובאיכות גבוהה, תוכניות עמידות בשינויים, שקל לשנותן בעת הצורך. ++C אינה המטרה, אלא **הכלי** להשגתה. כך למשל, אין להשתמש בפונקציות וירטואליות, אלא אם הן תומכות בפתרון בעיית התכנות.

התחלתי לעבוד בשפת ++C בינואר 1989 בסידני שבאוסטרליה, ומאז ועד כתיבת שורות אלו אני משתמש בה. למרות מספר השנים הרב בהם אני עוסק בתכנות מוכוון אובייקטים אני עדיין ממשיך ללמוד דברים חדשים לגבי ++C. באוסטרליה עבדנו על פרויקט שנחשב, בשעתו, מהפכני בסוגו. כאשר הציגו לנו את השפה חשבתי שזו טעות להשתמש בשפה חדשה בפרויקט חדש שבו יש טכנולוגיות חדשות ורמת הסיכון בו כה גבוהה. מאוחר יותר התברר ש- ++C עזרה לנו לפתח תוכנה באיכות גבוהה יותר ובזמן קצר יותר.

כיום אני עובד בחברת אורבוטק, ובה אנו מנצלים את השפה, ואת הטכניקות החדשות האלו, לפיתוח של פרויקטים מתקדמים. אני יכול להבטיח לקורא ש- ++C מספקת כלים יעילים מאוד למי **שיודע להשתמש** בהם! למי שאינו יודע להשתמש בכלי השפה, אייעץ שלא להשתמש בהם. כדאי לו, למשתמש, לנצל את אפשרויות השפה שהוא מכיר ובהן הוא שולט היטב.

קוראים המוצאים צורך בייעוץ, או מעוניינים לשאול שאלות, מוזמנים לפנות אלי. ניתן לעשות זאת בשתי דרכים: דרך **הוצאת הוד-עמי לספרי מחשבים** המוציאה לאור של ספר זה, או להשתמש בכתובת הדואר האלקטרוני שלי:

shimon@orbotech.co.il

# הדיסקט המצורף

**בדיסקט המצורף** לספר נמצאות דוגמאות תכנות רבות. כדי להדר את הדוגמאות, עליך להשתמש בפקודה make המסופקת עם **סביבת הפיתוח של בורלנד**. הדוגמאות **אינן תלויות במהדר מסוים**, אלא **מתאימות למהדרים רבים**. רבות מהדוגמאות הורצו גם על מהדר g++ שהוא חופשי לציבור. דוגמאות אחדות הורצו גם על המהדרים של חברת Sun. אם אתה משתמש במהדר אחר, עליך להחליף את פקודת ההידור על פי המהדר שברשותך. אם המהדר שברשותך תומך בפקודות make, עליך להחליף את הפקודות המתאימות בקובץ זה. **הוראות מפורטות להרצה ראה בנספח, וכמובן שעליך לעיין בהוראות ההפעלה של המהדר שברשותך**.

קובץ makefile הוא קובץ המורה למהדר כיצד יש להכין תוכניות לריצה. הישום make, מסופק עם המהדר, עובר על הקובץ makefile ומבצע את הפקודות שבו. בין הפקודות יכולות להיות פקודות אחרות מאשר להידור של קובצי תוכנית. בדרך כלל משתמשים ביישום זה כדי להכין תוכניות לריצה.

בתת הספרייה STL שבדיסקט המצורף לספר זה, תמצא את ספריית התבניות הסטנדרטיות של C++. זוהי הספרייה המקורית שסופקה על ידי חברת HP, והיא חופשית לקהל הרחב.

בהזדמנות זו, אני רוצה להודות לכל האנשים אשר עמדו לצידי בעת כתיבת ספר זה. להורי **רבקה וישראל כהן**, לאשתי **יפה** ולאחרונות החביבות והחשובות מאוד, בנותי: **ליאורה, נירית ואורנית**.

תודתי נתונה לכל האנשים שעזרו לי בעריכת ספר זה, במיוחד לצור לוין וליצחק עמיהוד.

שמעון כהן, 1997

# פרק 1

# סקירה של שפת C

בפרק זה נלמד את המבנים החשובים של שפת התכנות C. שפה זו היא הבסיס של ++C, ולכן חשוב להבינה. פרק זה נועד לסטודנטים, או מתכנתים שאינם מכירים את C ומשמש כמבוא לשפה, אשר מי שמכיר אותה יכול לפסוח עליו. הפרקים הבאים נכתבו בהנחת יסוד שהקורא מכיר את שפת C לפחות ברמה הבסיסית המוצגת בהמשך. תיאור נוסף של שפת C יכול הקורא למצוא באחד מהמקורות שבסוף הפרק: **המדריך השלם לשפת C** מאת משה ליכטמן ועמית רש, ו- **ללמוד C** מאת יואב נתיב, שניהם ב**הוצאת הוד-עמי**.

כל מה שנלמד בפרק זה על שפת התכנות C נכון גם לגבי ++C, שבנויה על יסודותיה של C ויורשת את תכונותיה. פרק זה משמש, אם כך, הקדמה לשאר הפרקים המלמדים ++C ותכנות מוכוון אובייקטים.

בעת הצורך נסקור יכולות המתאפשרות בעבודה עם ++C, אבל לא בעבודה עם C, כשיכולות אלה מתאימות למקרה ולדוגמה.

## 1.1 הקדמה והיסטוריה

**C** היא שפה רב תכלית אשר משופעת בביטויים, משפטי בקרה, מבני נתונים ופונקציות. זו שפה פונקציונלית שהתוכנית בה מורכבת מאוסף של פונקציות אשר קוראות זו לזו על מנת למלא משימה מסוימת. C אינה נחשבת לשפה גבוהה במיוחד. להיפך, C מאפשרת פעולות קרובות מאוד לחומרה של המחשב, ולכן היא גם שפה יעילה מאוד.

אחד הדברים המייחדים את שפת C הוא הפעולות על מצביעים ועל סיביות בודדות. המצביעים ב-C הם אולי הדבר שגורם ל-C להיות אחת השפות היעילות ביותר. אבל, מצד שני, המצביעים ב-C הם אולי הדבר המושמץ ביותר על ידי המתכנתים בשפות אחרות. המצביעים והיכולת לממש ביטויים מורכבים במשפט אחד, הם מן הדברים הקשים למתחילים ב-C. בהמשך נראה דוגמאות רבות לכך בספר זה.

במקור נועדה שפת C להיות שפת פיתוח למערכות הפעלה, עקב הקירבה שלה לחומרה מצד אחד ויעילותה מצד שני. יש מספר מערכות הפעלה שנכתבו ב-C, אשר המפורסמת מביניהן היא UNIX. ברבות הזמן הפכה C לשפה המובילה בין השפות הפונקציונליות, וכיום אפשר להבחין כי רוב התוכנות המצויות בשוק התוכנה נכתבו בשפה זו.

C פותחה על ידי **דניס ריצ׳י** ו**בריאן קרינגהן** מחברת AT&T ומומשה בתחילה עבור מערכת ההפעלה UNIX ומחשבים מסוג DEC ו-PDP-11. המהדר (מהדר), מערכת ההפעלה ומספר יישומי עזר נכתבו ב-C. שפה זו אינה קשורה כעת לחומרה מסוימת ואפשר להריץ תוכניות C על רבים מסוגי המחשבים שקיימים בעולם.

לשפת C יש יכולת גבוהה מאוד לבטא הרבה במספר מצומצם של משפטים. על כן, מתכנתים שאינם מכירים אותה לא ירגישו נוח עם יכולת זה. כדי לפתח תוכניות יעילות ב-C יש להכיר היטב את המשמעות של האופרטורים השונים ב-C.

ב-C יש **תהליך** בקרה יחיד, הקרוי **thread**, אשר מורכב מפונקציות, לולאות ומשתנים בסיסיים, כלומר C אינה תומכת באופן מובנה בפעולות מקבילות, כפי שעושה זאת שפת ADA, למשל. כדי להשוות, לדוגמה, בין שתי מחרוזות, צריך להפעיל פונקציה. לפיכך, C היא שפה קטנה ומצומצמת. זה נראה כחיסרון, אבל היתרון למתכנת הוא בכך שאפשר ללמוד C מהר מאוד ולהגיע לרמת שליטה גבוהה (דבר אשר אינו מדויק לגבי שפות אחרות, כמו ++C). למי שמכיר את התחביר (Syntex) של C אין קושי להבין מה עושה התוכנית. ++C אינה מכריחה מתכנתים להשתמש בדברים מורכבים אשר אינם קיימים ב-C.

## 1.2 תוכנית ראשונה

התוכנית הראשונה והבסיסית ביותר אשר ניתן לכתוב ב-C היא תוכנית המדפיסה הודעה לפלט. תוכנית כזו תדגים לנו כיצד משתמשים במהדר וכיצד מגדירים תוכנית אשר מקבלת פרמטרים בהפעלתה.

כל תוכנית בשפת C מתחילה בפונקציה מיוחדת ששמה **()main**. פונקציה זו היא הפונקציה הראשית של התוכנית וממנה מתבצעות התפצלויות וקריאות לפונקציות אחרות. שים לב שכותבים את שם הפונקציה ואחריה זוג סוגריים רגילים ריקים, או ״מלאים״ בארגומנטים (או פרמטרים). בכל תוכנית יכולה להיות, לכל היותר, פונקציה main אחת יחידה. כדי להדגים את הפונקציה הראשית נכתוב תוכנית קטנה אשר מדפיסה את הביטוי ״Hello World״ לפלט הסטנדרטי. בדרך כלל, הפלט הסטנדרטי הוא החלון, או המעטפת (shell), שממנו הופעלה התוכנית.

```
#include <stdio.h>
int main()
{
      printf("hello world\n");
      return 0;
}
```

המשפט הראשון בתוכנית שלנו (#include <stdio.h>) מכליל את הקובץ stdio.h לתוכנית. דבר זה נעשה כדי שהמהדר יכיר את הפונקציה ()printf, שהיא חלק מהספרייה הסטנדרטית של C. מייד לאחר מכן מוגדרת הפונקציה הראשית, ()main. הפונקציה הראשית היא, כאמור, פונקציה המייצגת את כל התוכנית והגדרתה נעשית בדומה להגדרת כל פונקציה ב-C. הגדרת פונקציה נפתחת בערך המוחזר על ידי הפונקציה (במקרה זה int), לאחר מכן שם הפונקציה ובסוגריים - רשימת הארגומנטים אשר הפונקציה מקבלת.

הארגומנטים אשר מועברים לפונקציה מועברים כהעתק של המשתנים המקוריים. המשתנים המקוריים מועתקים על המחסנית של הפונקציה, והפונקציה משתמשת בהעתק זה. שיטת העברת הארגומנטים נקראת **העברה לפי ערך** (by value). כדי לשנות את המשתנים המקוריים יש צורך להשתמש במצביעים (נראה כיצד לעשות זאת בסעיף העוסק במצביעים).

גוף הפונקציה נמצא בתוך סוגריים מסולסלים ({ }) ומהווה אוסף של משפטים אשר הפונקציה מבצעת. אוסף משפטים זה יכול לכלול הגדרת משתנים, או משפטי בקרה. פונקציה אף יכולה לכלול קריאה לפונקציות אחרות. כאשר הפונקציה מסתיימת היא **מחזירה ערך** בעזרת משפט **return**. המבנה הכללי של פונקציה הוא:

```
return-value func(arg-list)
{
      body
}
```

הפונקציה הראשית בדוגמה זו מחזירה ערך שלם (Integer) בעזרת המשפט return. הפונקציה הראשית אינה מקבלת ארגומנטים, בהמשך נראה שקיימים מצבים בהם מעבירים ארגומנטים לפונקציה הראשית. במקרה הנדון מחזירה הפונקציה את הערך אפס. הפונקציה מדפיסה לפלט את המחרוזת המבוקשת בעזרת קריאה לפונקציה אחרת, printf.

הפונקציה printf מדפיסה את המחרוזת הנתונה לפלט הסטנדרטי (stdio). הפונקציה יכולה גם להדפיס נתונים מעוצבים וזאת נראה בהמשך. התו המיוחד n\ (המיוצג בקוד על ידי שני תווים) מסמן למהדר להחליף אותו בקוד ASCII המתאים לקוד דילוג לשורה חדשה.

כדי להדר את התוכנית הזו צריך להפעיל את המהדר על הקובץ המתאים. פעולה זו נעשית כך:

```
cc hello.c
```

המהדר, התוכנית המבצעת את ההידור, יכולה להיות אחרת מאשר cc. עבור מהדרי Borland, למשל, שם התוכנית הוא bcc. ברירת המחדל של המהדר היא ליצור קובץ בשם a.out, שהוא למעשה התוכנית לאחר הידור, ואותו אפשר להריץ. כדי לשנות שם זה אפשר, בדרך כלל, להוסיף את הדגל (Flag) **o** (האות o), ולאחריו את שם הקובץ של התוכנית המבוקשת. למשל:

```
cc -o hello hello.c
```

פקודה זו תיצור קובץ בשם hello אשר יכיל את התוכנית לביצוע. כדי להפעיל את התוכנית צריך להקליד את שמה.

## 1.2.1 העברת ארגומנטים לתוכנית הראשית

שיפור אפשרי לתוכנית זו היא לאפשר הדפסת שם נתון. כלומר, לאפשר להדפיס hello, <name> כאשר <name> הוא ארגומנט שניתן על ידי המשתמש. כדי להזין את השם לתוכנית, יש להוסיף ארגומנטים לפונקציה main. דבר זה יכול להיעשות בצורה הבאה:

```
int main(int argc, char *argv[])
{
      printf("hello, %s\n", argv[1]);
      return 0;
}
```

הפונקציה הראשית של התוכנית נקראת **תמיד** עם שני ארגומנטים (הפרמטרים). הארגומנט הראשון הוא מסוג שלם, אשר מוגדר על ידי מילת המפתח int. ארגומנט זה מציין את מספר המחרוזות אשר הארגומנט השני מצביע עליהם. הארגומנט השני של הפונקציה מציין את הארגומנטים שאיתם הפעיל המשתמש את התוכנית.

הארגומנט השני הוא מערך של מצביעים לתווים. ראשית, הסוג של המערך הוא של תווים (char), כלומר, מערך של מצביעים לתווים. **המערך** (array) מסומן למהדר בעזרת הסוגריים המרובעים [], ואילו **מצביע** (pointer) מסומן למהדר בעזרת התו כוכבית *. משתנה מסוג תו הוא המשתנה הקטן ביותר וגודלו תלוי בסוג המחשב. ברוב המחשבים גודלו של משתנה זה הוא בית (byte) אחד, או שמונה סיביות.

הפונקציה printf יכולה לקבל מספר משתנה של ארגומנטים. הארגומנט הראשון הוא מחרוזת אשר מתארת את תסדיר (פורמט) הארגומנטים האחרים של הפונקציה. התסדיר של הארגומנטים נעשה בעזרת תווי הבקרה x%, כאשר x הוא התו שמתאר את סוג המספר הנדרש להדפסה. האות s מציינת שלפנינו מחרוזת (string), כלומר מצביע לתווים. הנה מספר תווי זיהוי:

- %s - הדפסה של מחרוזות.
- %d - הדפסה של שלמים.
- %f - הדפסה של מספרים ממשים בשיטת הנקודה הצפה (floating point).
- %l - הדפסה של שלמים כפולים באורכם (long).

הפונקציה printf מדפיסה את הפרמטרים שלה בתוך המחרוזת הנתונה לה לפלט הסטנדרטי של התוכנית. לכן התוכנית הקודמת תדפיס למשל:

```
hello, leora
```

## 1.2.2 הערות

כדי לאפשר תיעוד לתוכנה, מקובל להכניס הערות (comments) בגוף התוכניות. המהדר מתעלם מהערות אלה ואינו יוצר קוד כתוצאה מכך. הערות ב-C נכתבות בין שני זוגות של תווי קוד: /*... הערה ...*/. הערה מתחילה משמאל בתווים /* ומסתיימת מימין בתווים */. כאשר ההערה יכולה להיפרש על פני מספר רב של שורות. למשל:

```
/*
      The following function performs ...
*/
```

או כך:

```
/* הערה */
```

ב-++C יש צורה נוספת להערה: הערה בשורה אחת בלבד. ההערה מתחילה עם צמד התווים // בשמאל, ומסתיימת בסוף השורה. למשל:

```
// The following function performs ...
```

מניסיון, עדיף לשמור על סגנון הערות אחיד באותה תוכנית. למשל:

```
// avr computes the avarage of two numbers
int avg(int v1, int v2)
{
    // returns the average, possibly error when the sum
    // is odd
    return (v1+v2) / 2;
}
```

כדי לבטל קטע של פונקציה או את כולה, ניתן להכניסה להערה רבת שורות: /* .. */.

## 1.3 משתנים ופעולות אריתמטיות

ב-C יש סוגים שונים של משתנים (variables) שעליהם ניתן לבצע פעולות אריתמטיות כמו חיבור, חיסור, כפל וחילוק. כדי להגדיר משתנה, משתמשים במילת המפתח של סוג המשתנה ולאחריה מופיע שם המשתנה. למשל, השורה הבאה מגדירה משתנה מסוג שלם:

```
int i;
```

בשורה אחת ניתן להגדיר מספר משתנים. מפרידים ביניהם בפסיקים. למשל:

```
int i1, i2;
```

סוגי המשתנים השכיחים הם:

- char - מייצג תווים בשיטת ASCII. גודלו בית אחד.
- short - מייצג שלם קצר (int). הוא גדול או שווה ל-char.
- long - מייצג ערכים שלמים. כפול בגודלו מ-short.
- float - מייצג מספר ממשי עם שבר, למשל: 1.23.
- double - מספר ממשי בעל דיוק גבוה יותר מ-float.

את שלושת הסוגים הראשונים אפשר להגדיר חסרי סימן (unsigned), והמשמעות היא שהם יהיו חיוביים. לשם כך, צריך להוסיף להם את מילת המפתח unsigned. למשל:

```
int j;
unsigned k;
unsigned char c;
unsigned long l;
```

בשורה הראשונה מוגדר משתנה מסוג שלם **עם** סימן. השורה השנייה מגדירה משתנה שלם חסר סימן. **כאשר לא מסופק סוג המשתנה ברירת המחדל היא "שלם"**. בשורה השלישית מוגדר משתנה מסוג תו ללא סימן. בשורה הרביעית מוגדר משתנה שלם בגודל כפול ללא סימן.

במשתנה עם סימן יש סיבית אחת, אשר מציינת את הסימן (חיובי, או שלילי). לכן, הערכים שיכולים להחזיק מספרים חסרי סימן גדולים פי שתיים מאלה של משתנים עם סימן. למשל, משתנה מסוג char יכול לקבל ערכים בתחום [127, 128-], ואילו משתנה מסוג unsigned char יכול לקבל ערכים בתחום [255, 0].

באופן דומה ניתן להגדיר מספרים ממשים. השורה הבאה מגדירה שני משתנים מסוג מספר ממשי עם דיוק כפול:

```
double d1, d2;
```

על משתנים ניתן לבצע פעולות אריתמטיות. פעולות אריתמטיות נקראות **אופרטורים**. האופרטורים מתחלקים לשני סוגים:

- אופרטורים **בינאריים**, אשר פועלים על שני משתנים.
- אופרטורים **אונריים**, שפועלים על משתנה אחד.

רשימת הפעולות האריתמטיות:

| | |
|---|---|
| + | כדי לחבר שני מספרים, או כאופרטור אונרי לסמן את הסימן (כחיובי) של משתנה. |
| - | כדי לחסר משתנה ממשתנה אחר, או כאופרטור אונרי לסמן את הסימן (כשלילי) של משתנה. |
| * | אופרטור בינארי המשמש ככפל בין שני משתנים. |
| / | אופרטור בינארי המשמש לחלוקה של שני מספרים זה בזה. |
| % | אופרטור בינארי אשר מספק את השארית (מודולו) בפעולת חלוקה בין שני מספרים. |
| ++ | אופרטור אונרי המגדיל באחד את המשתנה שעליו הוא פועל. |
| -- | אופרטור אונרי המקטין באחד את המשתנה שעליו הוא פועל. |

למשל:

```
int i1 =1, i2 = 2, i3;
i3 = i1 * i2;
```

שורת ההגדרה מגדירה שלושה משתנים שלמים כאשר שני הראשונים מאותחלים ל-1 ו-2 בהתאמה ואילו השלישי אינו מאותחל. השורה הבאה היא שורת השמה, היא משימה את הערך של הכפל של המשתנים למשתנה i3.

כאשר משתמשים בפעולת החילוק, יש לדעת שעבור משתנים שלמים התוצאה מתעגלת ולכן:

```
i3 = i1 / i2;
```

משפט זה קובע למשתנה I3 את הערך אפס. לו המשתנים היו מסוג float או double היה הערך המוכנס ל-i3 שונה: 0.3333. **בפעולת החיסור יש להיזהר מחלוקה באפס**. מצב של חלוקה באפס אינו מוגדר ובמקרה הטוב, התוכנית תופסק באופן מיידי על ידי מערכת ההפעלה עם הודעה מתאימה. במקרה הפחות טוב, התוכנית לא תופסק ותתנהג באופן בלתי צפוי.

## 1.3.1 מערכים

ב-C וגם ב-++C יש אפשרות להגדיר מערכים. מערכים (arrays) הם אובייקטים אשר מכילים מספר קבוע מראש של אובייקטים אחרים. למשל מערך של תווים יכול להכיל עשרים תווים. מערך של שלמים יכול להעריך מספר שלמים. האובייקטים שמוגדרים במערך מאחסנים בקטע זיכרון רציף, מהאובייקט הראשון והלאה. כך אנו מגדירים מערך:

```
type name[size];
```

type מציין את סוג האובייקטים אשר נמצאים במערך. name הוא השם של המערך, ואילו הערך size בתוך סוגריים מרובעים, מציין את מספר האובייקטים שקיימים בו. למשל:

```
char array_of_chars[20];
int  array_of_ints[5];
```

המערך הראשון הוא מערך של 20 תווים ואילו המערך השני הוא מערך של עשרים שלמים. הגודל של המערך הראשון הוא עשרים תווים. הגודל של המערך השני הוא חמישה שלמים. הפנייה לאובייקטים במערך נעשית בעזרת פעולת המערך באופן הבא:

```
array_of_chars[0] = 'a';
array_of_int[1]   = 1;
```

מערכים מתחילים מאינדקס אפס ולכן, בשורה הראשונה מכניסים את הערך תו a לתו הראשון במערך, בשורה השנייה מכניסים את הערך 1 לכניסה השנייה במערך, אשר מצוינת על ידי האינדקס 1.

### 1.3.1.1 מערכים כפולים

מערכים כפולים הם מערכים של מערכים, ולעיתים מכונים **מטריצות** (Matrices). מערכים כפולים מוגדרים בצורה דומה למערכים רגילים כאשר מוסיפים מימד נוסף למערך. הצורה הכללית של הגדרת מערך כפול היא:

```
type name[size1][size2];
```

type מציין את סוג האובייקטים אשר יהיו במערך. name הוא שם המערך, size1 הוא גודל המימד הראשון של המערך, ואילו size2 הוא גודל המימד השני. למשל, אנו יכולים להגדיר מערך כפול של שלמים באופן הבא:

```
int array[10][5];
```

ההגדרה הזו מתייחסת למערך של 10x5 שלמים. המשמעות של ההגדרה היא: **מערך של מערכים**. כלומר, ההגדרה של המערך הכפול היא הגדרה של עשרה מערכים. כל אובייקט במערך הוא מערך של חמישה שלמים בפני עצמו. כלומר, בזיכרון יופיעו חמשת האובייקטים של המערך הראשון, אחריהם חמשת האובייקטים של המערך השני וכן הלאה. האובייקטים של המערך שבדוגמה הם שלמים.

פנייה לאובייקטים יחידים במערך נעשית כך:

```
array[1][2] = 3;
```

בדוגמה זו מוכנס הערך 3 לשורה השנייה של המערך במקום השלישי. במילים אחרות, הערך מוכנס לאיבר השלישי של המערך השני של מערכים.

מערכים מסדרים גבוהים יותר ניתנים להגדרה באופן דומה. לדוגמה, מערך תלת-מימדי של שלמים נראה כך:

```
int array[2][3][4];
```

הגדרה זו מגדירה מערך בגודל 2 של מערך כפול מגודל 3 של מערך שלמים מגודל 4. נוכל לומר זאת גם כך: לפנינו 3 מערכים של 4 איברים, או מערך דו-מימדי, אשר מוצג פעמיים ולכן יוצר מערך תלת-מימדי. במערך זה יש לנו 24 איברים שלמים (סה"כ 2*3*4). כלומר, ההגדרה הינה רקורסיבית.

## 1.3.2 דוגמה

כדי להמחיש משתנים ואופרטורים נכתוב תוכנית קטנה שמחשבת ממוצע של מספרים. נגדיר פונקציה שמקבלת מערך של שלמים ומספר האלמנטים במערך, ומחשבת את הממוצע שלהם. הפונקציה מחזירה את הממוצע שלהם לפונקציה הקוראת לה.

```
int avr(int arr[], int count)
{
	int i = 0, sum = 0;
	while (i < count) {
		sum = sum + arr[i];
		++i;
	}
	return (sum / count);
}
```

בפונקציה avr יש מספר דברים חדשים אשר לא ראינו עד כה. ראשית, היא מחזירה ערך מסוג שלם (int). הארגומנט הראשון שהפונקציה מקבלת הוא מערך של שלמים. הואיל והפונקציה אינה יודעת את גודל המערך (הניתן בארגומנט השני שלה) אין סימון של גודל המערך בתוך הסוגריים המרובעים שנלווים לשם המערך. המימד הראשון של המערך בקריאה לפונקציה הוא המימד היחיד אשר מותר לא לפרטו. המערך אינו מועבר בקריאה לפונקציה, אלא רק הכתובת לתחילתו. המהדר מחשב את האובייקט המתאים של המערך.

הפונקציה מגדירה שני משתנים i ו-sum אשר משמשים כאינדקס למערך וכסכום האיברים. שני המשתנים מאותחלים לאפס.

הפונקציה מחשבת את הסכום של איברי המערך בלולאת **while**. **לולאה** (loop) כזו מאפשרת לבצע רצף של מספר משפטים כתלות בתנאי. המבנה של לולאת while הוא:

```
while (expr) {
      body
}
```

לאחר מילת המפתח while מופיע ביטוי בין סוגריים. בכל שלב של הלולאה נבדק הביטוי. אם ערך הביטוי שונה מאפס, אזי מתבצע הגוף של משפט while. הגוף של משפט while נמצא בין סוגריים מסולסלים לאחר התנאי. גוף הלולאה יכול להכיל משפט אחד, או מספר משפטים. אם הגוף של משפט הלולאה מכיל משפט יחיד, אין צורך להכניסו בין סוגריים מסולסלים.

אם כך, המשמעות של משפט while היא, שכל עוד התנאי (הביטוי) בסוגריים הוא בעל ערך אמת (ערך השונה מאפס) מתבצע גוף ה-while. בכל פעם מתבצע גוף המשפט, והבקרה של התוכנית בודקת את התנאי במשפט האם לבצע את המשפט שוב. גם לפני הביצוע הראשון של משפט זה נבדק התנאי.

משפט while בפונקציה avr מתבצע כל עוד המשתנה i קטן מהמשתנה sum. האופרטורים > או < מקבילים למשמעות שלהם במתמטיקה, כלומר, קטן או גדול בהתאמה. האופרטורים =< ו-=> אף הם בעלי משמעות זהה למשמעות המתמטית שלהם, כלומר גדול-שווה או קטן-שווה. לפיכך, אופרטורים אלה מקבלים ערך אמת (ערך השונה מאפס, בדרך כלל אחד) כאשר המשתנים עליהם הם פועלים ממלאים את היחס המתמטי גדול או קטן בהתאמה. אחרת, האופרטורים מחזירים ערך שקרי (אפס). הסימן == ערך אמת, כל עוד המשתנים עליו הוא פועל בעלי ערכים זהים, אחרת אפס.

הגוף של הלולאה מתבצע count פעמים כאשר בכל פעם מתוסף הערך של האלמנט במקום ה-i במערך לסכום המערך. בסוף הלולאה המשתנה sum מכיל את הסכום של כל איברי המערך. לכן, אם מחלקים ערך זה במספר האלמנטים במערך מקבלים את הממוצע שלהם, וזה מה שהפונקציה מחזירה לפונקציה שקוראת לה.

המשפט sum = sum + arr[i] יכול להיכתב בצורה מקוצרת כך:

```
sum += arr[i];
```

המשמעות של המשפט המקוצר היא: הוסף למשתנה sum את הערך של האלמנט במקום ה-i של המערך. כאשר האופרטור צמוד לסימן =, אזי הפירוש שהאובייקט השמאלי עליו פועל האופרטור הוא האובייקט לפני =. למשל, עבור אופרטור בינארי @ כלשהו מתקיימת הזהות הבאה:

```
x @= y;  <==>  x = x @ y;
```

לכן על פי חוק זה מתקיימות הזהויות הבאות:

```
x -= y;  <==>  x = x - y;
x *= y;  <==>  x = x * y;
x /= y;  <==>  x = x / y;
```

## 1.3.3 מצביעים

ב-C יש משתנים אשר מצביעים (pointers) על משתנים אחרים. כלומר, משתנים שמכילים כתובת של משתנה אחר. לפיכך, הערך של משתנה כזה הוא כתובת בזיכרון. האופרטור * משמש להגדיר מצביעים לסוג נתון (פרט לכך שהוא אופרטור הכפל). כאשר מוסיפים לו את סוג המשתנה, הרי שההגדרה היא של מצביע לסוג מסוים. למשל:

```
int *ip;
```

המשפט לעיל מגדיר מצביע למשתנים מסוג שלמים. כאשר המהדר מטפל במשתנה כזה הוא מניח שהוא מצביע לשלם ובהתאם לכך הוא מבצע עליו פעולות אריתמטיות. אם למשל משתנה מסוג שלם הוא בגודל ארבעה בתים אז המשפט - `++ip;` יקדם את הכתובת אליו מצביע המשתנה בארבע ולא באחד. לעומת זאת הפעלת אותו אופרטור על משתנה מסוג מצביע לתו תשנה את הכתובת (ערכו) בבית אחד.

כאשר מפעילים את האופרטור * על מצביע, מקבלים את הערך של האלמנט אשר מוצבע על ידי המצביע, כלומר, את הערך של האלמנט (המשתנה) שנמצא בכתובת שמכיל המצביע. לפעולה זו קוראים dereference.

כדי להכניס ערך למשתנה מסוג מצביע יש לתת לו את הכתובת של משתנה המתאים בסוגו לסוג המצביע. למשל, עבור מצביע לשלמים יש לתת את הכתובת של מצביע שלם. לקיחת הכתובת של משתנה נעשית על ידי האופרטור &. למשל,

```
ip = &i;
```

כאשר i הוא משתנה מסוג שלם (int). אפשר להשתמש במצביע כדי לשנות ערכים של משתנים אשר מוצבעים על ידי המצביע. למשל, אפשר לעדכן את הערך של המשתנה i בעזרת האופרטור כוכבית ( *). דבר זה ניתן לעשות בצורה הבאה:

```
*ip = 3;
```

המשפט הזה מכניס את הערך 3 למשתנה i. באופן דומה אפשר לקבל את הערך של המשתנה i. לפיכך, מצביעים מאפשרים לנו לבצע פעולות על משתנים בצורה עקיפה.

```
int j = *ip;
```

במקרה זה אנו מגדירים משתנה שלם j אשר מאותחל בערך התכולה של המצביע ip שהיא הערך של המשתנה I.

### 1.3.3.1 מצביעים ומערכים

בשפת C, מצביעים ומערכים קשורים זה לזה והמשמעות שלהם היא כמעט זהה. כאשר נתון מערך, השם של המערך הוא מצביע לתחילת המערך. מצביעים ומערכים זהים ואפשר להשתמש ולהחליף ביניהם. ההבדל העקרוני הגדול בין מערך לבין מצביע הוא, ששם המערך מצביע למקום קבוע בזיכרון ואי אפשר לשנותו כך שיצביע למקום אחר. לעומת זאת, מצביע אפשר לשנות ולגרום לו להצביע למקום אחר בזיכרון.

האריתמטיקה אשר פועלת על מערכים פועלת על מצביעים כדי לחשב ערכים של אובייקטים בתוך מערך. למשל:

```
int array[10], *ap;
ap = array;
array[2] = 3;
ap[2] = 3;
```

בקטע הקוד הזה מוגדרים מערך בגודל של עשרה שלמים ומצביע לשלמים. ההגדרה של מערך גורמת להקצאה של עשרה שלמים כאשר השם (משתנה) array מצביע לתחילת המערך. לעומת זאת, ההגדרה של המשתנה ap גורמת להגדרת מצביע בלבד. משפט ההשמה אשר מופיע לאחר מכן נותן למצביע ap להצביע לתחילת המערך. שני המשפטים שמופיעים לאחר מכן מכניסים את הערך 3 לאלמנט השלישי של המערך. האינדקס אשר מופיע בפעולת המערך הוא 2, אבל מערכים ב-C **מתחילים מאפס** ולכן זהו האלמנט השלישי.

כאשר המהדר מחשב כניסה במערך הוא עורך את החישוב יחסית למצביעים. כלומר, הכתובת של תחילת המערך נלקחת כבסיס לחישוב. לכתובת זו מוסיפים את האינדקס מוכפל בגודל בבתים של אלמנט במערך, וזו הכתובת של האלמנט שיש להתייחס אליו. לאחר חישוב הכתובת מתבצעת פעולת dereference, כלומר הוצאת התכולה מהכתובת, ועל הערך המתקבל מתבצעת הפעולה המבוקשת. למשל:

```
array[2] = 3;  <==> *(array + 2 * sizeof(int)) = 3;
```

לסיכום, יש שני הבדלים עיקריים בין מערך למצביע:

- בהגדרת מערך, שם המערך הוא מצביע קבוע לתחילת המערך ואין אפשרות לשנות אותו. מצביעים יכולים להצביע למקומות שונים, כי הם יכולים לקבל ערכים שונים במהלך התוכנית.
- כאשר מגדירים מערך, המהדר מקצה עבורו זיכרון לפי מספר האלמנטים שנמצאים בו. כאשר מגדירים מצביע, המהדר מקצה מקום אך ורק עבור המצביע.

לכן, הפונקציה avr יכולה להיכתב בצורה הבאה:

```
float avr(int *arr, int count)
{
      int i = 0, sum = 0;
      while (i < count) {
            sum += arr[i];
```

```
            ++i;
      }
      return ((float)sum / (float)count);
}
```

השינוי שהוספנו כאן הוא שהערך המוחזר מהפונקציה הוא float, ואילו הפונקציה מקבלת מצביע לתחילת מערך השלמים. הסיבה להמרה של שלם למספר ממשי היא, שכאשר מחלקים מספרים שלמים מקבלים מספרים ממשיים (לא תמיד שלמים). בגירסה הקודמת היינו מאבדים דיוק.

כדי למנוע אבדן דיוק, אנו מורים למהדר לבצע את פעולות החישוב במספרים ממשיים על ידי המרות (casting) של המשתנים במשפט return. המרה של משתנה נראית כך:

```
x = (type)y;
```

כאן, type הוא הסוג של המשתנה x. בדרך זו אנו ממירים את הערך של y לסוג המתאים ל-x.

### 1.3.3.2 שינוי הארגומנטים של פונקציה

כזכור, כשקוראים לפונקציה עם ארגומנטים ב-C (וגם ב-C++), מועבר העתק של המשתנה אל המחסנית של התוכנית. הפונקציה מכירה רק את ההעתק, וכל פעולה על ההעתק **אינה משפיעה** על המשתנה המקורי. לעיתים צריך לשנות את המשתנים המקוריים. נניח למשל, שישנה פונקציה שמחליפה בין הערכים של שני משתנים שלמים באופן הבא:

```
void swap(int v1, int v2)
{
      int tmp = v1;
      v1 = v2;
      v2 = tmp;
}
```

פונקציה כזו יכולה לשמש אלגוריתם של מיון למשל. הבעיה היא שקריאה לפונקציה כזו אינה משפיעה על המשתנים המקוריים, כלומר:

```
int x1 = 1, int x2 = 2;
swap(x1, x2);
```

גם לאחר הקריאה לפונקציה, המשתנים x1 ו-x2 יחזיקו את הערכים ההתחלתיים שלהם, כלומר 1 ו-2 בהתאמה. אנו יכולים לשנות מצב זה אם נגדיר את הארגומנטים של הפונקציה כמצביעים. במקרה כזה, יועברו הכתובות של המשתנים המקוריים ואז הפונקציה תוכל להשפיע על תוכן המצביעים:

```
void swap(int *v1, int *v2)
{
      int tmp = *v1;
      *v1 = *v2;
      *v2 = tmp;
```

```
}

int x1 = 1, x2 = 2;
swap(&x1, &x2);
```

לאחר הקריאה לפונקציה swap במקרה זה, יוחלפו הערכים של המשתנים המקורים x1 ו-x2. השימוש במצביעים כדי להשיג מטרה זו נראה מגושם ואכן, ++C תומכת באפשרות אחרת שנקראת **ייחוס** (reference) כדי להשיג את אותה תוצאה. נדון בכך בפרקים הבאים.

### 1.3.3.3 מצביעים לפונקציות

ב-C וב-++C אפשר להגדיר מצביעים לפונקציות. בניגוד לשם הפונקציה שקשור לכתובת אחת ויחידה של פונקציה, מצביע לפונקציה אינו קשור לכתובת אחת. לפיכך, אפשר לשנות את המצביע, כך שיצביע לפונקציה אחרת. מכיון שאפשר להפעיל את הפונקציה הקשורה באותה עת למצביע, הרי שאפשר לשנות את הפונקציה המופעלת בזמן ריצה. ב-C דבר זה מעניק גמישות רבה לתוכניות, אך ב-++C יש כלים הרבה יותר חזקים, ובדרך כלל אין צורך להשתמש באפשרות זו.

הגדרת מצביע לפונקציה כלשהי נראית בצורתה הכללית כך:

```
ret-value (*ptr)(arg-list);
```

כשקוראים את ההגדרה משמאל לימין, מוגדר תחילה הערך שהפונקציות מחזירות, שהמצביע אמור להצביע עליהם. לאחר מכן, בתוך סוגריים, מוגדר שם המצביע לאחר התו *. לאחר שם המצביע מוגדרים סוגי הפרמטרים שמועברים לפונקציה.

כדי להפעיל פונקציה דרך מצביע, יש לספק את הארגומנטים ולבצע את פעולת "התוכן" על המצביע באופן הבא:

```
(*ptr)(a1, a2, ..., an);
```

כדי להבהיר נושא זה נציג דוגמה פשוטה. נגדיר שתי פונקציות, אחת מחשבת סכום של זוג מספרים, והשנייה - מחשבת את המכפלה שלהם:

```
int sum(int x1, int x2)
{
      return (x1 + x2);
}
int mult(int x1, int x2)
{
      return (x1 * x2);
}

int x1, x2;
int (*func)(int, int); // a pointer to a function which
                       // takes two numbers and returns one
```

```
func = sum;
x1 = (*func)(1, 2);    // x1 <-- 3
func = mult;
x2 = (*func)(1, 2);    // x2 <-- 2
```

דוגמה זו מראה מצד אחד את הגמישות אשר מושגת בעזרת מצביעים לפונקציות, ומצד שני קטע הקוד אינו ברור. לא תמיד ברור לאן מצביע המצביע לפונקציה, כלומר, לא ברור תמיד איזו פונקציה תתבצע כאשר מפעילים פונקציות באמצעות מצביעים. בדוגמה זו ההשמה למצביע נעשית בסמוך להפעלת הפונקציה. לכן, מצב זה עדיין קל יחסית. ייתכן מצב בו ערך המצביע נקבע במקום אחר בקוד מאשר המקום שבו מופעלת הפונקציה!

## 1.3.4 אופרטורים להוספה והפחתה

ראינו שיש אופרטורים המאפשרים להגדיל, או להקטין משתנה ביחידה אחת. האופרטור ++ מאפשר להגדיל משתנה ביחידה אחת, בשעה שהאופרטור -- מאפשר להקטין משתנה ביחידה אחת. אופרטורים אלה ניתנים להפעלה בשתי דרכים:

```
int I;
I++;
++I;
I--;
--I;
```

שני האופרטורים הראשונים מגדילים את המשתנה I באחד, ואילו שני האחרונים מקטינים את המשתנה I באחד. אמנם, האופרטורים מבצעים פעולות זהות - אבל **יש משמעות והבדל בסדר הפעולות.**

כאשר כותבים את האופרטור ++ מימין לשם המשתנה, כמו למשל I++, משתמשים תחילה בערך של המשתנה I ולאחר מכן מגדילים את ערכו באחד. פעולה כזאת היא **פעולה מאוחרת**: **postfix**, כלומר שינוי ערך המשתנה מתרחש **לאחר** השימוש בו. לעומת זאת, כאשר כותבים את האופרטור ++ משמאל, כמו למשל ++I, מגדילים תחילה את ערך המשתנה I ורק לאחר מכן משתמשים בו (בערך החדש). זוהי **פעולה מוקדמת**: **prefix**. חשוב להבחין בדרך שבה משתמשים בערך של המשתנה, לפני או אחרי שינויו, כמו למשל כאשר מעבירים אותו למשתנה אחר. למשל:

```
int j, k;
j = 1;
k = ++j;     k <-- 2
k = j++;     k <-- 2
```

בקטע התוכנית הנ"ל המשתנה j מקבל את הערך 1. לאחר מכן המשתנה k מקבל את הערך של המשתנה j לאחר הגדלתו, כלומר k מקבל את הערך 2. בשורה הבאה מקבל k את הערך של j ולאחר מכן מוגדל המשתנה j. לכן, בסוף קטע זה, הערך של k הוא 2 ואילו הערך של j הוא 3.

## 1.3.5 ביטויים לוגיים

במשפטי בקרה (שנלמד בהמשך) יש בדרך כלל **תנאי לוגי** (logical condition), אשר נבדק וכתוצאה מכך מתבצע חלק זה או אחר של משפט הבקרה. התנאי הלוגי הוא ביטוי של השפה, אשר מתפתח לערך השונה מאפס, **אמת** - ערך 1 (**true**), או **שקר** - ערך 0 (**false**). למשל, ביטוי מהצורה:

```
x > 5
```

מציג ערך לוגי 1 אם המשתנה x מכיל ערך הגדול מ- 5, אחרת הוא מציג ערך 0. כל פעולת השוואה מציגה בסיומה ערך לוגי, כפי שנראה להלן:

| | |
|---|---|
| x > y | הביטוי מקבל ערך אמת אם הערך של המשתנה x גדול מהערך של y, אחרת ערך 0. |
| x < y | הביטוי מקבל ערך אמת אם הערך של x קטן מהערך של y, אחרת ערך 0. |
| x >= y | הביטוי מקבל ערך אמת אם הערך של x גדול או שווה לערך של y, אחרת 0. |
| x <= y | הביטוי מקבל ערך אמת אם הערך של x קטן או שווה מהערך של y, אחרת 0. |
| x == y | הביטוי מקבל ערך אמת אם הערך של x שווה לערך של y, אחרת 0. |
| x != y | הביטוי מקבל ערך אמת אם הערכים של המשתנים x ו-y שונים, אחרת 0. |

השוואות אלו פועלות על כל משתני השפה. כאשר המשתנים הם מצביעים, ההשוואות פועלות על הכתובת, ולא על תוכן המצביע.

אפשר לצרף מספר ביטויים כאלה ולקבל ביטוי מורכב יותר. צירוף הביטויים יכול להיעשות בעזרת האופרטורים הבאים:

- && - שקול **ל-and** לוגי ("**גם**", "**וגם**"), ומקבל ערך השונה מאפס, אם שני הביטויים עליהם הוא פועל שונים מ- 0.
- || - שקול ל-or לוגי ("**או**"), ומקבל ערך השונה מאפס, אם אחד מהביטויים עליו הוא פועל שונה מ- 0.

למשל, כאשר ברצוננו לבדוק אם הערך של משתנה נמצא בין חמש לשבע, אנו יכולים לרשום את הביטוי הבא:

```
x >= 5 && x <= 7
```

הביטוי הזה מציג ערך אמת (1) רק אם המשתנה x נמצא בתחום המבוקש, אחרת הוא מציג ערך false. לעומת זאת, אם אנו רוצים לבדוק אם המשתנה הזה אינו נמצא בתחום, נכתוב את הביטוי הבא:

```
x < 5 || x > 7
```

מספיק שאחד מתת-הביטויים מקבל את הערך 1, אז הביטוי הכולל מקבל את הערך 1. בצורה זו אפשר להרכיב ממספר ביטויים ביטוי מורכב יותר. אין הגבלה של השפה על מספר תת-הביטויים הנכתבים בביטוי.

האופרטור "**!**" (not, "**לא**") הופך את הערך של הביטוי מ- 0 ל- 1 ולהיפך. כלומר, האופרטור הופך את הביטוי מערך אמת לערך שקר. לכן שני הביטויים הבאים שקולים:

```
!(x < 5 || x > 7) <==> (x >= 5 && x <= 7)
```

## 1.3.6 אופרטורים הפועלים על סיביות

C היא שפה יעילה המאפשרת לפעול על סיביות בודדות של מילת מחשב. בדרך כלל, פעולות אלו לא נתמכות בשפות עיליות אחרות וכדי לממשן כותבים את הביטויים האלה באסמבלר (שפת מכונה). יש שני סוגים של פעולות כאלה: **פעולות לוגיות** (logical operations) על סיביות ו**פעולות הזזה** (shift operations) של סיביות.

### 1.3.6.1 פעולות לוגיות על סיביות

ביטויים לוגיים על סיביות מאפשרים לקבל ערך של סיבית, או לשנות ערך של סיבית. יש שני אופרטורים המבצעים פעולת and או or על משתנה:

& - מבצע פעולת **and** ("**וגם**", או "**גם**") ברמת הסיביות בין שני ביטויים.

| - מבצע פעולת **or** ("**או**") ברמת הסיביות בין שני ביטויים.

למשל:

```
int x1, x2, x3, x4;
x1 = 1;
x2 = 2;
x3 = x1 & x2;
x4 = x1 | x4;
```

במשתנה x1 יש את הערך1...0 ואילו במשתנה x2 יש את הערך 10...00, ולכן לאחר פעולת or מקבל המשתנה x4 את הערך 011...0 ואילו x3 את הערך 0...0.

אופרטור נוסף הוא האופרטור ^ אשר מבצע פעולת **xor** (exclusive or, "זה או זה בלבד") על שני משתנים או ביטויים. התוצאה של פעולה זו היא ברמה של סיביות. אם הסיביות של שני הביטויים הנתונים לאופרטור דולקות או כבויות, תהיה הסיבית המתאימה בתוצאה כבויה. אם רק סיבית אחת (ורק אחת) דלוקה בשני הביטויים הנתונים לאופרטור דלוק, תהיה הסיבית המתאימה בתוצאה דולקת. למשל:

```
int x1 = 3, x2 = 1;
int x3 = x1 ^ x2;
```

המשתנה x1 מכיל את הערך 00...11 ואילו המשתנה x2 מכיל את הערך הבינארי 00...10, ולכן התוצאה של הפעלת האופרטור האחרון תהיה 00...01. במילים אחרות, הערך של x3 יהיה אחד.

### 1.3.6.2 פעולות הזזה על סיביות

אפשר להזיז את הסיביות במשתנים שלמים מסוג: long, short, char, int. דבר זה נכון גם למשתנים מהסוגים הקודמים ללא סימן, כלומר, למשתנים מסוג unsigned. ההזזות של ערכים אלו מתבצעות על ידי שני אופרטורים:

- >> - הזזה ימינה של הביטוי הנתון.
- << - הזזה שמאלה של הביטוי הנתון.

למשל:

```
int x1, x2, x3;
x1 = 1;
x2 = x1 << 1;
x3 = 8;
x2 = x3 >> 2;
```

המשתנה x2 מקבל את הערך של המשתנה x1 (שהוא במקור 0001 בינארי) לאחר שערך זה הוזז פעם אחת שמאלה. לכן, בשורת ההשמה הראשונה של x2 יקבל משתנה זה את הערך 2 (כלומר - 0010 בינארי). בשורת ההשמה השנייה של המשתנה x2 מוכנס אליו הערך של x3 (שהוא 8, או 1000) לאחר הזזה ימינה פעמים (שתוצאתה 0010). לפיכך, בסוף קטע הקוד הזה, המשתנה x2 מכיל את הערך 2.

## 1.3.7 מחרוזות

מערכים מיוחדים ב-C הם מחרוזות. מחרוזות הם מערכים של **תווים (char)**. למשל מערך של מערך של תווים יכול להיראות כך:

```
char *str = "abcd";
```

המחרוזת "abcd" היא כתובת בזיכרון של מערך של תווים. כאשר כותבים ביטוי כזה, המהדר דואג לסיים את המחרוזת תמיד בערך מיוחד, אשר מסמן את סוף המחרוזת. ערך זה הוא **אפס**. לפיכך, מערך של תווים דומה למערך של שלמים, פרט לעובדה שיש לו אלמנט נוסף שמסמן את סופו והוא הערך אפס. אפשר היה להגדיר את המערך הזה בצורה הבאה:

```
char str[] = "abcd";
```

עתה str הוא שם של מערך בגודל חמישה תווים, כאשר התו הנוסף הוא התו "אפס" בסוף המערך. המהדר מקצה את הזיכרון בגודל המתאים למערך זה באופן אוטומטי.

## 1.4 משפטי בקרה

משפטי בקרה מאפשרים לנו כתיבת לולאה. בשפת C יש משפטי בקרה מסוגים שונים, כפי שנראה להלן.

### 1.4.1 משפט הלולאה while

משפט הלולאה while (**"כל עוד..."**) מאפשר להגדיר **לולאה** (loop) פשוטה. היא פועלת, כלומר המשפט או המשפטים הכלולים בה מתבצעים, כל עוד מתמלא התנאי הכתוב בפקודה. נבחן את הפקודה באמצעות דוגמה של העתקת מחרוזת.

נכתוב פונקציה אשר מעתיקה מערך של תווים אחד למשנהו.

```
void strcpy(char *dst, char *src)
{
      int j;
      j = 0;
      while (src[j]) {
            dst[j] = src[j];
            ++j;
      }
      dst[j] = 0;
}
```

הפונקציה strcpy מקבלת שני מצביעים לתחילת המערכים ומחזירה ערך **void**. מילת המפתח void אינה מציינת ערך מסוים ואין משתנים מסוג זה. במקרה זה מילת המפתח void מציינת שהפונקציה אינה מחזירה כל ערך.

המצביע הראשון (dst) מתייחס למערך שאליו יש להעתיק את המערך שהמצביע השני (src) מתייחס אליו. הפונקציה עושה שימוש בכך שהמערך של המקור חייב להסתיים באפס. בלולאה **while** היא מעתיקה את תוכן המערך אל המערך המבוקש, עד לתו אפס (זהו תנאי משתמע מהגדרת המחרוזת, שבסופה כתוב הערך "0"). לאחר הלולאה הפונקציה כותבת אפס בתו האחרון שבמערך המטרה.

הפונקציה strcpy בגירסה הנוכחית מכילה משפט שמגדיל את המשתנה j. משפט זה יכול להיכנס למשפט ההשמה ואז הפונקציה תיראה כך:

```
void strcpy(char *dst, char *src)
{
      int j;
      j = 0;
      while (src[j])
            dst[j] = src[j++];
      dst[j] = 0;
}
```

אנו מגדילים את j לאחר השימוש בערכו. לפיכך, המהדר מחשב תחילה את הכתובת של האלמנט j במערך src, והוא משתמש בערך זה כדי להשים את הנתון למערך dst במקום ה-j. הבעיה בגירסה זו של הפונקציה היא שתחילה (במקרים מסוימים) מבוצע הצד הימני של משפט ההשמה, ולאחר מכן הצד השמאלי. לכן, ההשמה תתבצע לאלמנט הבא של dst. ב-C אין הגדרה קבועה לסדר של משפט ההשמה.

דרך אחרת לכתוב את הפונקציה הזו היא להשתמש באריתמטיקה של מצביעים. נכתוב את הפונקציה בצורה הבאה:

```
void strcpy(char *dst, char *src)
{
      while (*src) {
            *dst = *src;
            ++dst;
            ++src;
      }
      *dst = 0;
}
```

בגרסת המצביעים אנו מקדמים את המצביעים במקום להגדיר משתנה אינדקס למערך ולקדם אותו. למרות שבגירסה זו יש יותר משפטים, היא הרבה יותר מהירה מהגירסה הקודמת, מכיון שכאשר משתמשים באינדקסים למערך יש לבצע פעולת חיבור בכל פנייה עם אופרטור המערך []. לכן, משפט כמו src[j]=dst[j] מחייב חיבור של הערך j לכתובות של תחילת שני המערכים. בנוסף, היה עלינו להגדיל את המשתנה j בגירסה הקודמת. בגירסה זו יש לנו רק שתי פעולות אריתמטיות, ואילו בגירסה הקודמת היו לנו שלוש.

ניתן לקצר את הפונקציה הזו בצורה יותר קומפקטית באופן הבא:

```
void strcpy(char *dst, char *src)
{
      while (*dst++ = *src++) ;
}
```

הפונקציה צומצמה לשורה אחת בלבד. בתנאי של משפט while מתרחשת ההשמה של התו הנוכחי ששני המצביעים מצביעים עליו, ולאחר מכן מקודמים המצביעים. הערך של משפט ההשמה הוא התו האחרון אשר הושם. אם הערך הוא אפס, כלומר, סוף המחרוזת - אזי הסתיימה ההעתקה.

למרות שהגירסה הנוכחית של הפונקציה קצרה מהגירסה הקודמת, היא פחות יעילה. הגירסה הנוכחית משתמשת באופרטור ++ בצורה שבה נלקח הערך של המשתנה ורק אז מוגדל המשתנה. לכן, על המהדר לזכור את הערך של המשתנה במשתנה זמני, לקדם את המצביע ואז להשים את הערך הזמני למקום המתאים לפי מצביע המטרה (dst). ההשמה למשתנה הזמני דורשת זמן, ולכן הגירסה הנוכחית איטית יותר בביצוע מהגירסה הקודמת.

## 1.4.2 משפט התניה if

משפט התניה ב-C מורכב לפחות ממילת המפתח **if** (״**אם**״) ולעיתים עם מילת המפתח **else** (״**אם לא**״). הצורה הכללית של משפט הבקרה נראה כך:

```
if (cond1) {
      stat1;
}
else if (cond2) {
      stat2;
}
...
else {
      statn;
}
```

אפשר להסתכל על משפט הבקרה if כמורכב משני חלקים עיקריים: חלק של if וחלק אופציונלי של else. החלקים האופציונליים יכולים להיות מורכבים ממספר חלקים של else. כאשר החלקים הפנימיים של else חייבים להיות מסוג **if-else**. רק המשפט האחרון יכול להכיל else בלבד.

משפט בקרה זה מבוצע מהתחלה שלו (if-part) והלאה. אם נמצא תנאי שמתפתח לערך אמת, ההסתעפות המתאימה של המשפט מבוצעת. אם אין שום הסתעפות שהתנאי שלה שונה מאפס ויש חלק else - מבוצע חלק זה. אם אין חלק כזה, לא מתבצע שום חלק של המשפט. להלן מספר דוגמאות:

```
int x;
x = 1;
if (x > 0)
      printf("x is positive");
```

בדוגמה זו נבדק ערכו של x. זה התנאי של משפט הבקרה, הואיל והתנאי מקבל ערך אמת (כי x גדול מאפס) יופעל הגוף של משפט הבקרה (printf). במקרה שיש משפט יחיד, אין צורך להכניס את הגוף בסוגריים מסולסלים. אפשר להרחיב את הדוגמה באופן הבא:

```
if (x > 0)
      printf("x is positive");
else
      printf("x is not positive");
```

בצורה זו של משפט הבקרה יש לנו חלק אחד של else, שמתבצע רק אם החלק של if אינו מתבצע; כלומר, אם התנאי של if מקבל ערך שקרי (אפס). במקרה שלנו, אם הביטוי x>0 הוא לא נכון יתבצע החלק שאחרי else.

אפשר להוסיף למשפט הזה else נוסף בצורה הבאה:

```
if (x > 0)
      printf("x is positive");
```

```
else if (x == 0)
      printf("x is zero");
else
      printf("x is negative");
```

החלק האחרון (else) מתבצע, בתנאי שהמשתנה x אינו חיובי ואינו שווה לאפס. כלומר, החלק האחרון מתבצע אם המשתנה x הוא בעל ערך שלילי.

צורה אחרת למשפט התניה היא בעזרת האופרטור **?** ("**האם**"). הצורה הכללית של אופרטור זה היא:

```
(cond ? stat1 : stat2);
```

משפט זה מקבל את הערך של stat1 אם הערך של cond שונה מאפס, ואחרת את הערך של stat2. למשל:

```
int x1, x2, x3;
...
x3 = (x1 < x2 ? x1 : x2);
```

בדוגמה זו המשתנה x3 מקבל את הערך המינימלי של המשתנים x1 ו-x2. אם x1 קטן מ-x2 תוצאת האופרטור **?** היא x1, אחרת x2.

## 1.4.3 משפט for

גם משפט הוא משפט לולאה (loop). משפט זה מאפשר לבצע משפט אחד, או יותר, מספר רב של פעמים. הצורה הכללית של המשפט היא:

```
for (init; condition; inc) {
      body;
}
```

לאחר מילת המפתח for מופיעים בתוך סוגריים רגילים שלושה משפטים. המשפט הראשון הוא אתחול שמבוצע פעם אחת ויחידה במהלך הלולאה. בדרך כלל מאתחלים במשפט זה משתנה שהוא אינדקס, או מצביע לערך התחלתי. המשפט השני מייצג את התנאי לביצוע הגוף של משפט הלולאה. אם התנאי מקבל ערך השונה מאפס (ערך אמת הוא בדרך כלל אחד), מתבצע הגוף של המשפט אשר מופיע בסוגריים מסולסלים. החלק השלישי הוא משפט אשר מקדם אינדקס בדרך כלל, או מצביע, ועקב כך יכול לשנות את התנאי של המשפט. החלק השלישי מתבצע כל פעם לאחר ביצוע גוף משפט הלולאה.

הגוף של משפט הלולאה מופיע בסוגריים מסולסלים ומורכב ממשפט ביצוע אחד, או יותר. אם קיים רק משפט ביצוע אחד, אין צורך בסוגריים המסולסלים. משפטי ביצוע אלה יכולים להיות גם משפטי בקרה אחרים, ואין הגבלה איזה משפטים נכתוב, או בכמה רמות קינון נשתמש.

משפט הלולאה for שקול למשפט הלולאה while הזה:

```
init;
while (condition) {
      body;
      inc;
}
```

נדגים את משפט הלולאה for על ידי חישוב הממוצע, כמו שעשינו באחד הסעיפים הקודמים.

```
float avr(int *arr, int count)
{
      int i, sum = 0;
      for(i = 0; i < count; ++i)
            sum += arr[i];

      return ((float)sum / (float)count);
}
```

בדוגמה זו מאותחל האינדקס של הלולאה i לאפס בחלק האתחול של משפט הלולאה. התנאי של הלולאה בודק אם האינדקס קטן ממספר האלמנטים במערך. משפט ההגדלה של האינדקס מגדיל את המשתנה i בכל פעם שהלולאה מבוצעת, והוא עושה זאת לאחר ביצוע גוף הלולאה. לכן, לאחר הפעם הראשונה שמתבצע גוף הלולאה (sum += arr[i]) יקבל המשתנה i את הערך אחד (1).

## 1.4.4 משפט do-while

משפט **do-while** (**"בצע כל עוד..."**) הוא משפט לולאה שדומה למשפט while אך סדר הפעולות וההתניה שונים בו. המבנה הכללי של המשפט:

```
do {
      statements;
} while (expr);
```

תחילה מוצגת מילת המפתח **do**, ולאחר מכן מופיעים משפטים לביצוע בין סוגריים מסולסלים. אם יש משפט ביצוע יחיד, אין צורך בסוגריים המסולסלים. לאחר הסוגריים מופיעה מילת המפתח **while** ולאחריו בין סוגריים ביטוי אשר מקבל ערך אמת או שקר.

גוף המשפט (המשפטים בין הסוגריים המסולסלים) מתבצעים לפחות פעם אחת וממשיכים להתבצע כל עוד התנאי של הביטוי שלאחר ה-while הוא **ערך אמת** (true). לכן משפט זה זהה למשפט while, פרט לכך שהלולאה מתבצעת לפחות פעם אחת. בגוף המשפט יכולים להופיע משפטים אחרים, כולל משפטי בקרה, ואין מגבלה לסוג המשפטים שנכתבים או למספר המשפטים המקוננים.

## 1.4.5 משפט switch

משפט בקרה **switch** (״**מַתֵג**״, או ״**מיתוג**״) מאפשר לבחור בין מספר אפשרויות של ערכים שלמים ולבצע משפטים של C בהתאם לבחירה. המבנה הכללי של משפט הוא:

```
switch  (int-val) {
      case val1:
            stat1;
            break;
      case val2:
            stat2;
            break;
      ...
      default:
            statn;
            break;
}
```

משפט switch הוא משפט בקרה הסתעפותי רב שלבי. לאחר מילת המפתח **switch** נכתב בין סוגריים משתנה שמכיל ערך שלם. משתנה שלם (int) הוא משתנה מסוג: long, char, int או כל אחד מאלה בצירוף מילת המפתח unsigned. הגוף של המשפט שנמצא בין סוגריים מסולסלים מכיל מספר משפטי **case** ואפשרות למשפט **default** אחד. המשפט **default** הוא משפט **ברירת המחדל** שמתבצע אם כל שאר האופציות אינן בעלות ערך אמת. אין חשיבות לסדר ההופעה של המקרים (case) וברירת המחדל (default).

לאחר כל מילת מפתח **case** (״**אירוע**״ או ״**מקרה**״) מופיע ערך שלם, אשר מציין מתי לבצע את המשפטים שאחריו. כלומר, **אם** הערך שעליו מבצעים את משפט switch זהה לערך של משפט case (הערך שכתוב לאחר מילת המפתח case) מתבצע משפט זה. בסוף כל משפט case נהוג לכתוב **break**. מילת המפתח האחרונה מסמנת למהדר שאין צורך לבצע את המקרה (case) הבא. אם מילת מפתח זו חסרה, יבוצע משפט case הבא אחריו. לפיכך, מתבצע משפט case אשר הערך שלו זהה לערך של המשתנה הנוכחי שעליו מבצעים את switch.

משפט זה שקול למשפטים הבאים:

```
if (int-val == val1) {
      stat1;
}
else if (int-val == val2) {
      stat2;
}
...
else {
      statn;
}
```

למשל אנו יכולים לכתוב את קטע הקוד הבא:

```
int x;
switch (x > 0) {
      case 1:
            printf("x is positive\n");
            break;
      case 0:
            printf("x is not positive\n");
            break;
      default:
            printf("could not happen\n");
            break;
}
```

התנאי x>0 יכול להתפתח לשני ערכים 1 או 0. אם הביטוי x>0 מתפתח ל-1, אז מתבצע המקרה הראשון case 1. לעומת זאת, אם הביטוי מתפתח לאפס, מתבצע המקרה השני case 0. המקרה האחרון שהוא ברירת המחדל (default) אינו מתבצע בדוגמה זו. כאשר האפשרויות לבחירה הן שלמים, אזי משפט switch הוא האפשרות המתאימה ועדיפה על פני משפט הבקרה if-else.

## 1.5 רשומות

**רשומות** הן **מבנים** (structures) הן הכלי ששפת C מספקת כדי לאגד מספר סוגי משתנים במבנה אחד. כאשר יש אובייקטים, או ישויות, במרחב הבעיה אשר מייצגים נושא אחד, אפשר לאגדם ברשומה אחת. המבנה הכללי של רשומה הוא:

```
struct Name {
      memebers
};
```

הרשומה מתחילה במילת המפתח **struct** ("**מבנה**"), ולאחר מכן מופיע שם הרשומה. שם הרשומה הוא אופציונלי ואינו חייב להופיע. אם אין שם לרשומה, אי אפשר להגדיר לה מופעים נוספים (מאוחר יותר נראה כיצד אפשר לעשות זאת). לאחר שם הרשומה מופיעים המבנים אשר שייכים לה. בין מבנים אלה יכולים להופיע רשומות אחרות. תכונה זו מאפשרת לבנות רשומות מסובכות וגדולות. אין מגבלה על מספר תת-הרשומות, או למידת הקינון שלהן.

דוגמה לרשומה המייצגת נקודה המכילה קואורדינטות x ו-y:

```
struct Point {
      int x, y;
};
```

הרשומה שהוגדרה כאן היא בעלת השם Point. שני המבנים ברשומה הם מסוג שלמים. הצהרה זו אינה מקצה זיכרון או משתנים, אלא רק מוסיפה סמל לטבלת הסמלים של המהדר. אפשר להגדיר משתנה מסוג הרשומה בדרך זו:

```
struct Point pnt;
pnt.x = 5;
pnt.y = 6;
```

בדוגמה הגדרנו משתנה pnt מסוג Point, בצורה זו יש להוסיף את מילת המפתח struct. בשתי השורות שלאחר מכן מוכנסים ערכים 5 ו-6 לשדות x ו-y בהתאמה. הפנייה לשדות הרשומה נעשית על ידי האופרטור ".", המפריד בין שם הרשומה לשם השדה שפונים אליו. אם יש קינון של מספר רשומות, הפנייה לשדה ברשומה תיעשה כך:

```
struct X x;
x.m1.m2.m3 = 3;
```

כלומר, מפעילים את האופרטור "." מספר פעמים לפי עומק השדה שאליו רוצים להגיע. ב-++C אין צורך להוסיף את מילת המפתח Struct וניתן לעשות זאת בצורה הבאה:

```
Point pnt;
```

כדי להגדיר מצביע לרשומה ניתן לעשות זאת בצורה הבאה:

```
struct Point *pntptr;
```

או ב-++C אפשר להגדיר זאת ללא מילת המפתח struct בצורה הבאה:

```
Point *pntptr;
pntptr = &pnt;
ptr->x = 1;
```

הכנסת הערכים או פנייה לשדות במקרה שיש מצביע לרשומה נעשית על ידי האופרטור -> שמבצע פעולה זהה לאופרטור "." על רשומות. בדוגמה השדה x מקבל את הערך 1.

## 1.5.1 שימוש ב-typedef

בעזרת משפט **typedef** (**"הגדר סוג"**) ניתן להגדיר שם חדש לסוג מסוים. בצורה זו אפשר לקצר ולחסוך כתיבה של שמות. הצורה הכללית של משפט זה היא כדלהלן:

```
typedef type-name new-name;
```

המשפט מתחיל במילת המפתח **typedef**, לאחר מכן מופיע שם הסוג של המשתנה שעבורו מגדירים שם חדש. לבסוף מופיע השם החדש של אותו סוג. למשל:

```
typedef unsigned int uint;
typedef unsigned char uchar;
```

בשתי השורות האלו הגדרנו שני שמות חדשים לסוגים בסיסיים אשר קיימים בשפה, uint ו-uchar. הראשון מביניהם מתאים לסוג שלם ללא סימן, ואילו השני מתאים לסוג תו ללא סימן. לאחר הגדרות אלו אנו יכולים להגדיר את המשתנים הבאים:

```
uint I1;
unsigned int I2;
unsigned char c1;
uchar c2;
```

המשתנים I1 ו-I2 הם מאותו הסוג, שלם ללא סימן (unsigned). המשתנים c1 ו-c2 הם מאותו סוג, תו ללא סימן. בדומה אפשר להגדיר שם חדש לרשומה שמייצרת נקודה, כך:

```
typedef struct {
      int x, y;
} Point;

Point p1, *pp1;
pp1 = &p1;
```

במשפטים אלה הגדרנו משתנה מסוג Point בשם p1. בנוסף, הגדרנו משתנה המצביע ל-Point בשם pp1.

כדוגמה לרשומה יותר מורכבת, אנו יכולים להגדיר רשומה המייצגת קו ישר. כידוע, קו ישר מוגדר על ידי שתי נקודות, ולכן אנו יכולים להשתמש בהגדרות הקודמות של נקודה כדי להגדיר קו ישר באופן הבא:

```
struct Line {
      Point p1;          // first point
      Point p2;          // second point
};

Line l1;
l1.p1.x = 0;
l1.p1.y = 0;
l1.p2.x = 5;
l1.p2.y = 6;
```

בדוגמה זו הגדרנו ישר אשר עובר דרך ראשית הצירים ודרך הנקודה (5,6). כדי להכניס ערכים אלה לשדות המתאימים, יש להשתמש בשם השדה המתאים לאחר הנקודה. כך למשל, שורת ההשמה הראשונה כותבת את ערך אפס (0) לשדה x של הנקודה הראשונה בקו.

## 1.6 פונקציות רקורסיביות

פונקציות רקורסיביות (recursive functions) הן פונקציות אשר קוראות לעצמן באופן ישיר, או בלתי ישיר. פונקציות רקורסיביות יעילות במספר הגדרות מתמטיות, או באלגוריתמים של מיון. למשל, ההגדרה המתמטית של עצרת (!) היא:

```
n! = n * (n -1)!
1! = 1
0! = 1
```

בצורה זו אפשר לכתוב פונקציה רקורסיבית שמחשבת עצרת של מספר כלשהו.

```
int factorial(int n)
{
      if (n ==0 || n == 1)
            return 1;
      else
            return n * factorial(n - 1);
}
```

הפונקציה שמחשבת את ערך העצרת במקרה זה אינה יעילה, מכיון שהיא מבצעת n-1 קריאות לפונקציות. אפשר לתקן את הפונקציה בצורה הבאה:

```
int factorial(int n)
{
      int res = 1, k;
      for (k = 2; k <= n; ++k)
            res = res * k;
      return res;
}
```

בגירסה השנייה, הפונקציה אינה קוראת לפונקציות אחרות (או לעצמה, במקרה הרקורסיבי), אלא מבצעת את החישוב בעזרת לולאה. לא תמיד אפשר לפתור בקלות כה רבה בעיה כזו, ואז יש לכתוב פונקציה רקורסיבית.

כאשר נקראת פונקציה, המשתנים בפונקציה נכתבים במחסנית של התוכנית, וגם הפרמטרים הנמסרים לפונקציה מועתקים אל המחסנית. אין כל קשר בין המשתנים של הפונקציה הקוראת לאלה של הפונקציה הנקראת. תכונה זו מאפשרת לבצע קריאות רקורסיביות לפונקציות.

## 1.7 דוגמאות לקלט/פלט בסיסי

קלט/פלט בסיסי כולל את האפשרות לקלוט תו אחד, או לפלוט תו אחד. בסעיף זה נראה קלט/פלט בסיסי כזה ומספר תוכניות שימושיות שמאפשרות תהליכים פשוטים ושימושיים. שתי הפונקציות הבסיסיות שישמשו אותנו כאן הן:

```
int getchar();
void putchar(char);
```

הפונקציה getchar קוראת את התו המגיע מהקלט ומחזירה את ערכו. אם הפונקציה מגיעה לסוף הקובץ, מוחזר EOF שהוא סמל קבוע המוגדר על ידי משפט define. משפט define מוכר על ידי קדם המהדר אשר מופעל לפני המהדר, מבטל את ההערות ומבצע מספר מבני בקרה. נחזור לנושא זה בהמשך פרק זה.

בכל הדוגמאות שלהלן נשתמש ביכולת של מערכות הפעלה כמו UNIX לנתב (redirect) את הפלט, או את הקלט של התוכנית, בעזרת האופרטורים < או > של מעטפת (shell) מערכת ההפעלה. כלומר, אם מפעילים תוכנית p כלשהי בצורה הבאה:

```
p < in > out
```

הקלט הסטנדרטי של התוכנית ינותב לקובץ in, הפלט ינותב לקובץ out.

## 1.7.1 ספירת תווים בקובץ נתון

הדוגמה הראשונה תהיה של תוכנית הסופרת את מספר התווים בקובץ נתון, ומדפיסה את מספר התווים.

```
int main()
{
	int nchars = 0, c;
	while ((c = getchar()) != EOF)
		++nchars;
	printf("file has %d chars\n", nchars);
	return 0;
}
```

התוכנית קוראת את התווים מהקלט הסטנדרטי, תו אחר תו, עד שנקרא ערך המסמן את סוף הקובץ. הפונקציה getchar מחזירה שלם, מכיון שיש ערך נוסף, פרט לערכים הרגילים של תווים, המסמן את סוף הקובץ. כלומר, מכיון שרוצים לקרוא את כל סוגי התווים, הפונקציה צריכה להחזיר ערך השונה משאר התווים שמסמן את סוף הקובץ.

## 1.7.2 העתקת קבצים

אפשר לשכלל את הדוגמה הקודמת, בכך שנעתיק קובץ כלשהו לתוך קובץ אחר ונחזיר את מספר הבתים בקובץ המקור. גם בתוכנית זו אנו קוראים תו אחר תו מהפלט הסטנדרטי, וכותבים תו אחר תו לפלט הסטנדרטי. כאמור, אנו משתמשים ביכולת של מערכות הפעלה כמו UNIX, DOS או Windows, לנתב את הפלט, או את הקלט של התוכנית אל קבצים. לכן, הפעלה של התוכנית הבאה בצורה copy<input>output מנתבת את הפלט של התוכנית לקובץ input ואת הפלט לקובץ output.

```
int main()
{
	int nchars = 0, c;
	while ((c = getchar()) != EOF) {
```

```
            ++nchars;
            putchar(c);
        }
        return nchars;
}
```

תוכנית זו, שלא כמו התוכנית הקודמת, אינה מדפיסה את מספר התווים לפלט הסטנדרטי. אם התוכנית היתה מדפיסה את מספר התווים, אזי דבר זה היה מודפס לקובץ המטרה.

תוכנית זו כמו התוכנית הקודמת אינה יעילה, משום שהקריאה לפונקציה עבור כל תו (char). אנו מדגימים את השימוש במבני השפה, ולכן איננו שמים דגש על יעילות. בפרקים הבאים נראה דוגמאות רבות לשימוש יעיל בכלים המסופקים על ידי ++C.

## 1.7.3 ספירת בתים ושורות

הדוגמה הבאה היא של תוכנית הסופרת את מספר התווים ואת מספר השורות בקובץ. כדי לספור את מספר השורות, יש להשוות את התו הנקרא לתו המייצג שורה חדשה. אם תו זה אכן זהה לתו המייצג שורה חדשה יש להגדיל מונה.

```
int main()
{
  int nchars = 0, nlines = 0, c;
  while ((c = getchar()) != EOF) {
      ++nchars;
      if (c == '\n')
            ++nlines;
  }
  printf("file has %d lines and %d chars\n", nlines,
nchars);
  return 0;

}
```

כדי לקבל תווים מיוחדים, כמו התו המייצג שורה חדשה, יש להקדימו בתו \ (לוכסן הפוך) כאשר שני התווים נמצאים בין גרשיים בודדים מסוג '. יש מספר תווי בקרה שימושיים שמפורטים כדלהלן:

- '\n' - שורה חדשה (new line).
- '\t' - טבלר (tab) סדרה של מספר תווים רקים, בדרך כלל שמונה.
- c% - לכל תו c, הביטוי הזה מחזיר את הערך ASCII שלו שהוא בין 0 ל-256.

## 1.8 קדם-מהדר

ב-C וגם ב-++C יש שלב הקודם להידור. בשלב זה מופעל **קדם מהדר** (pre-processor, שהוא רכיב של המהדר עצמו), אשר סורק את קובץ הקלט ומבצע משפטים המתחילים בתו #. כל משפט כזה מסתיים בסוף השורה. אם יש צורך להמשיך משפט בקרה כזה מעבר לשורה אחת, יש להשתמש בתו \ לפני סוף השורה. המעבד מבטל את ההערות אשר נמצאות בגוף התוכנית ומאחד את הקבצים שנכללו בקבצים אחרים. רק לאחר מכן, מועבר הקוד המעובד למהדר.

### 1.8.1 הכללת קבצים

הכללת קבצים נעשית על ידי מילת הבקרה **include**, אשר נכתבת לאחר התו המיוחד המציין את פקודות הקדם המהדר:

```
#include <filename>
#include "filename"
```

באפשרות הראשונה שם הקובץ להכללה נמצא בין <>. במקרה זה הקובץ מחפש לפי הדגלים I-(i מציין דגל כלשהו), אשר הועברו למהדר. למשל:

```
cc -Lc:\inc
```

פקודה זו (עם הדגל L-) מוסיפה את הספרייה c:\inc למסלול החיפוש של קבצים. במקרה זה, קדם המעבד מחפש את קובץ נדרש בספרייה הנוכחית, ורק אם לא מצא אותו הוא עובר לחפש בספריות שמוגדרות בנוסף.

### 1.8.2 הגדרות של קבועים

משפט אחר של הקדם מעבד הוא **define** ("**הגדר**") שמאפשר להגדיר קבועים, או טקסט להחלפה. הצורה הכללית של משפט זה היא:

```
#define name val
```

לאחר הגדרה זו, המעבד עובר על קובצי המקור ובכל מקום בו הוא מוצא את המילה name הוא מחליף במילה val. למשל, נהוג להגדיר את הקבועים הבאים באופן הבא:

```
#define TRUE 1
#define FALSE 0
```

הקדם מעבד תומך במשפט בקרה אחר המאפשר **לבדוק הגדרה** (על ידי **ifdef**), כלומר, אם שם כלשהו כבר הוגדר:

```
#ifdef TRUE
...
#else
...
#endif
```

מבנה הפקודה דומה למבנה של כל פקודת if או if-else. לרוב משתמשים פקודה ifdef כדי למנוע הכללה של קבצים פעמיים. למשל,

```
#ifndef FILE_H
#define FILE_H

void fun(int x, int y);

#endif
```

המשפט ifndef כולל בתוכו את ההמשך, אם השם המופיע לאחריו אינו מוגדר. במקרה זה תוכן הקובץ יועבר למהדר. אם השם הוגדר כבר, תוכן הקובץ אינו מועבר למהדר.

### 1.8.3 הגדרת תבנית פונקציה

קדם המהדר מאפשר להגדיר תבנית פונקציה. אין זו פונקציה ממש אלא תבנית שנראית כמו פונקציה. כלומר, אם מופיע שם ולאחריו רשימת ארגומנטים בין סוגריים ומופרדים בפסיקים, המהדר מתייחס למשפט זה כאל פונקציה. קדם המהדר במקרה זה מחליף את השם של הפונקציה בגוף שלה בכל מקום בקוד, לאחר החלפת הארגומנטים של תבנית הפונקציה. למשל:

```
#define min(x, y) ((x) < (y) ? (x) : (y))

int j = 1, k =2;
int z = min(j, k);
```

המשפט אשר משתמש במילת המפתח min מתורגם על ידי קדם המהדר בצורה הבאה:

```
int z = ((j) < (k) ? (j) : (k));
```

משתמשים באמצעי זה כדי להימנע מקריאה לפונקציות קצרות ולחסוך בזמן. מומלץ לא להשתמש ביכולת זו של C, משום שב-++C יש אמצעים אחרים וטובים יותר שנראה בהמשך.

## 1.9 סיכום

בפרק זה למדנו את המבנים הבסיסיים של C. כאשר היה שוני בין מבנה כלשהו של C לבין המבנה ב-++C הדגשנו את השוני. על כן, בפרק זה למדנו גם את המבנים של ++C המאפשרים תכנות פונקציונלי. **תכנות פונקציונלי** (functional programming) מבוסס על פונקציות, שאחת מהן היא פונקצית ניהול ראשית שמחלקת את העבודה בין פונקציות שונות. כל פונקציה אחרת יכולה לחלק את העבודה לפונקציות אחרות שהיא קוראת להן לצורך כך.

ב-C (וגם ב-++C) יש משתנים בסיסיים של השפה. משתנים אלה מתאימים לסוגים שונים כמו תווים (char), שלמים קצרים (short), רגילים (int) וארוכים (long). C כוללת משתנים ממשיים מדיוק רגיל (float), או דיוק כפול (double).

C מאפשרת להגדיר מבני רשומות (struct) אשר מאגדות מספר משתנים בסיסיים של השפה, או רשומות אחרות. אפשר להשתמש ברשומות כדי לייצג ישויות ממרחב הבעיה שאנו מנסים לפתור.

פונקציות ב-C יכולות לקבל מספר פרמטרים ולהחזיר תשובה אחת. הפרמטרים שמועברים ב-C הם על ידי ערך. כלומר, הפרמטר מועתק למחסנית של הפונקציה הנקראת. לכן, אם הפונקציה משנה את הפרמטר הנמסר לה, אין זה משפיע על המשתנה המקורי. תכונה זו מאפשרת לקרוא לפונקציות באופן רקורסיבי. כלומר, פונקציה יכולה לקרוא לעצמה באופן ישיר, או עקיף. כדי לשנות את הערך המועבר לה על הפונקציה לקבל מצביע.

בפרקים הבאים נתרכז במבנים של ++C המאפשרים תכנות מונחה אובייקטים, כמו מחלקות ירושה וקשירה דינמית של פונקציות לאובייקטים. כל מה שלמדנו בפרק זה תקף עבור ++C והינו הבסיס של שפה זו.

## 1.10 שאלות

1. כתוב פונקציה המחשבת ממוצע של מערך מספרים בעזרת משפט for.
2. כתוב את הפונקציה strlen(char *s) אשר מחזירה את אורך המערך של התווים שהיא מקבלת.
3. כתוב פונקציה רקורסיבית הממיינת מערך של שלמים באופן הבא:

   היא ממיינת את שני חצאי המערך על ידי קריאה רקורסיבית לעצמה, ולאחר מכן היא ממזגת את שני החצאים הממוינים.
4. כתוב תוכנית אשר מדפיסה לפלט את מספר השורות בקובץ נתון, מספר המילים ומספר התווים (תוכנית זו מתנהגת כמו wc ב-UNIX).
5. כתוב פונקציה שמחשבת את האיבר n-י של הסדרה הבאה:

   An = An-1 + An - 2 כאשר A0 = 0 ואילו A1 = 1.

## 1.11 מקור

The C Programming Language Brian W.Kernighan, Dennis M. Ritchie, Prentice Hall.

פרק 2

# תכנות מוכוון אובייקטים

בפרק זה נלמד מהו **תכנות מוכוון אובייקטים** (Object Oriented Programming), או **OOP**. נלמד על חסרונות התכנות הפונקציונלי, שהיה ועדיין מקובל כשיטת תכנות ליישומים רבים. נתאר את ההתפתחות מהשיטות הפרוצדורליות לתכנות מוכוון אובייקטים ונראה כיצד תכנות מוכוון אובייקטים עונה לחסרונות אלו. כאן המקום להדגיש שתכנות מוכוון אובייקטים ויישומו בשפת ++C **אינם** הפתרון האוטומטי לכל התחלואים אותם נלמד. יש להשתמש בתבונה ב-++C, וגם בתכנות מוכוון אובייקטים, כדי להגיע לתוצאות הרצויות.

## 2.1 משבר התוכנה

בשנות השבעים החלו לפתח תוכנות גדולות יותר. מחשבים מהירים יותר, ושפות תכנות פרוצדוראליות, הם שהקנו למתכנתים עוצמה רבה. עם זאת, מרבית הפרויקטים לא עמדו בלוח הזמנים שלהם, ואלה שעמדו בלוח הזמנים, הפיקו מוצרי תוכנה באיכות נמוכה מהצפוי.

האיכות הנמוכה נבעה מהעובדה שכאשר המתכנתים נדרשו לערוך שינויים בתוכנה, הם נאלצו להשקיע מאמצים רבים כדי להשיג את מטרתם. יתר על כן, ברבות הזמן וכתוצאה מהשינויים האלה, הפכה התוכנה למסובכת והיה קשה מאוד לשנותה. כאשר הצליחו לשנות את התוכנה, גרמו השינויים לכשלים (או באגים) במקומות לא צפויים בתוכנה.

מחקרים שנעשו באותה תקופה, גילו שמירב ההשקעה בתוכנה לא היה בזמן הפיתוח, אלא בזמן התחזוקה. מתכנתים "מבלים" זמן רב בניסיון לשיפור התוכנה והוספת אפשרויות חדשות בה. כתוצאה מפעילות זו מתגלים כשלים חדשים לתוכנה הקיימת. במילים אחרות, התוכנה נהפכת למסובכת כתוצאה מאפשרויות שמוסיפים לה לאחר התכנון הראשוני.

מהנדסי תוכנה שניסו לשפר את המצב, הגיעו למסקנה שחלקי התוכנה השונים תלויים זה בזה, ולכן שינוי בחלק אחד של התוכנה גורם לשינוי בחלקים אחרים. הבנה זו הביאה לעולם את ה**מודולים**.

## 2.2 מה הם מודולים

**מודולים** (modules) הם קטע תכנות, לרוב אוסף של פונקציות, שמטפלות בנושא מסוים מוגדר, כמו לדוגמה מודול המטפל ברשימה. רק הפונקציות המסוימות של המודול יכולות להכניס **קישורים** (links) חדשים, או לבטל קישורים קיימים, ורק הן מכירות את מבנה הרשימה. כתוצאה מכך, כאשר משתנה מבנה הרשימה - לא מושפעת המערכת כולה מן השינוי.

### 2.2.1 דוגמה: ספר טלפונים - ניסיון ראשון

כדי להמחיש את הנושא נראה דוגמה. נלמד תחילה איך נכתב הקוד הדרוש בשפת C, ומאוחר יותר נלמד כיצד ניתן לממשה ב-++C.

נניח שיש לנו תוכנית אשר מַחזקת רשימה של אנשים ומספרי הטלפון שלהם. אפשר להוסיף שם ומספר טלפון ואפשר להציג את רשימת הטלפונים, או את רשימת האנשים. לצורך כך נגדיר **מבנה** (structure) אשר יכיל שם ומספר טלפון:

```
typedef struct node_t
{
      int phone_number;
      char name[32];
}
node;
```

תחילה נבנה את הרשימה **כמערך** (array) של המבנה, ונגדיר מספר מקסימלי של אלמנטים למערך. נגדיר את הערך המקסימלי כ-MAX_PHONES ואת מספר האלמנטים הנוכחים במערך כמשתנה הגלובלי current_used.

```
#define MAX_PHONES 100
int current_used;
node phones[MAX_PHONES];
```

כעת נגדיר פונקציה שמציגה את רשימת האנשים:

```
void print_people()
{
      int i;
      for (i=0; i<current_used; i++)
            printf("name = %$\n", phones[i].name);
}
```

הפונקציה עוברת על המערך עד למשתנה current_used האחרון ומדפיסה את השם השמור בכל רשומה. באופן דומה, אפשר להגדיר פונקציה שמדפיסה את מספרי הטלפון בלבד:

```
void print_phones()
{
      int i;
      for (i=0; i<current_used; i++)
            printf("name = %d\n", phones[i].phone_number);
}
```

לצורך הוספת אנשים ומספרי טלפון, צריכה להיות לנו פונקציה נוספת שנקרא לה add_phone. פונקציה זו מקבלת שם ומספר טלפון ומכניסה אותו למערך הרשומות.

```
void add_phone(char *name, int num)
{
      phones[current_used].phone_number = num;
      strcpy(phones[current_used++].name, name);
}
```

במהלך הזמן יקרה לתוכנית הקטנה שלנו מה שקורה לפרויקטים גדולים: הרחבות כתוצאה מדרישות נוספות, או דרישות שלא הובנו היטב מלכתחילה. למשל, ברבות הזמן ניצור פונקציות נוספות אשר יציגו בפנינו את מספרי הטלפון של כל האנשים בחתך מסוים, טלפונים באזור חיוג מסוים וכדומה. כל אותן הפונקציות פועלות על המערך המקורי של מספרי הטלפונים, אבל אז יכול לקרות מה שקורה בדרך כלל בפרויקטים גדולים, נרצה לשנות את מבנה הנתונים.

קיימים מספרי טלפון רבים, למעשה ללא גבול נראה לעין. לכן, נרצה לעבור למבנה של **רשימה מקושרת** (linked list) אשר אין לה גודל קבוע (בניגוד למבנה שיצרנו המוגבל ל-100 מספרים). רשימה מקושרת היא רשימה המורכבת מתאים שכל אחד מהם מצביע לבא אחריו בתור, ולכל תא יש את הנתון שלו. אפשר לעבור על רשימה מקושרת בעזרת מעקב על המצביעים של התאים אשר מקשרים בין תאים שכנים ברשימה. בהמשך נראה דוגמה לרשימה כזו. אך מעבר זה דורש מאיתנו לעבור על הפונקציות שכתבנו ולשנותן. בעצם, דורש מעבר שכזה לכתוב את התוכנה מחדש!

כיצד, אם כן, יכולנו למנוע זאת בשיטת המודולים?

## 2.2.2 בניית ספר טלפונים בעזרת מודולים

כדי למנוע מכל הפונקציות שלנו את הכרת מבנה הנתונים המשמש להחזקת מספרי טלפון, נגדיר **פונקציות שירות** (service functions), אשר יתנו לה שירותים בסיסיים על מבנה הנתונים. מבנה הנתונים עצמו יהיה מוסתר מכל שאר חלקי התוכנה. כלומר, פונקציות אחרות שלא שייכות למודול יגשו למבנה הנתונים דרך פונקציות השירות של המודול בלבד.

למשל, נגדיר פונקציות שירות המאפשרות לעבור על השמות ומספרי הטלפון שלנו זה אחר זה, באופן הבא:

```
int current = 0;
node *first_node()
{
      current = 0;
      return (phones + current);
}

node *next_node()
{
      current++;
      return (phones + current);
}
```

כדי לעבור על רשימת הטלפונים ולהציג את מספרי הטלפונים, עלינו לכתוב פונקציה שמשתמשת בפונקציה first_node ובפונקציה next_node באופן הבא:

```
void print_people()
{
      for (node *nptr = first_node(); nptr; nptr =
next_node())
            printf("\t name = %$\n", nptr->name);
}
```

כעת, אם נרצה לעבור למבנה נתונים אחר, למשל **רשימה** (list), נשנה את שלוש הפונקציות המטפלות במבנה הנתונים של הטלפונים בלבד, ולא נאלץ לכתוב את כל התוכנית מחדש. כלומר, הפונקציה שמדפיסה את ספר הטלפונים אינה משתנה! בדרך זו, הפונקציות שמשתמשות במודול ואינן שייכות למודול, אינן משתנות כתוצאה משינוי במבנה הנתונים של המודול. בדרך זו, של מידור התוכנה והפיכתה למודולים, אנו מגינים על חלקי התוכנה מפני שינויים מסיביים.

בדרך שתוארה עד כה יש חיסרון בולט: אפשר להגדיר רשימה אחת ויחידה בכל התוכנית. בשלב הבא, נרצה לתמוך במספר רשימות כאלה להתייחס לרשימה כמודל אוטונומי המטפל באוסף של רשומות.

## 2.2.3 טיפול במספר מודולים

כפי שראינו, טיפול במודול אחד בו-זמנית מספיק לתוכניות פשוטות ביותר. במציאות, צריך לפעול בו-זמנית במספר מודולים.

אפשר, אם כן, להגדיר מודול המטפל ברשימות כלשהן. מודול זה יתן לנו שירותים, כגון הוספת אלמנטים לרשימה או ביטולם. כמו כן, המודול יספק את האפשרות לעבור על הרשימה, כלומר, לטייל על הרשימה מהאלמנט הראשון לשני וכו׳. פעולת המעבר,

או הסריקה, על הרשימה נקראת **איטרציה** (iteration), ומכאן נגזר המונח **איטרטור** (iterator), שאודותיו נלמד בהמשך. כאשר נזדקק לרשימה של אלמנטים נבקש עבורה **ידית** (handle) שתשמש, לאחר מכן, בכל הקריאות לפונקציות של מודול זה.

נגדיר את הממשק למשתמש בקובץ הנקרא **קובץ ממשק** (interface file), בו נתאר את הפעולות שמספק המודול. קובץ הממשק נקרא, בדרך כלל, **קובץ כותר** (header) של מודול. הקובץ מפרט את הפעולות אשר בהן תומך המודול.

```
struct list;
struct list *list_create();
void list_add(struc list *l,void *data);
void *list_first(struct list*);
void *list_next(struct list*);
```

השורה הראשונה מצהירה על מבנה (רשומה) בשם list. שורה זו אינה מפרטת את מבנה המבנה (או הרשומה. לעיתים מבנה נקרא גם רשומה והכוונה זהה). לאחר הצהרה על רשומה בדרך זו, אפשר להשתמש **במצביע** (pointer) לרשומה בלבד. אי אפשר להגדיר **משתנה** מסוג רשומה משום שהמהדר אינו מכיר את השדות שלה. הגדרה זו מספקת **ידית** לרשומה, ומסתירה את הפרטים מהמשתמש ברשומה.

ההצהרה על פונקציות הטיפול ברשומה משתמשת בידית של הרשומה כדי לפעול על הרשומה המבוקשת. חשובה במיוחד הפונקציה list_create, שיוצרת רשומה חדשה ומחזירה מצביע לרשומה זו. המצביע לרשומה הוא הידית שעל המשתמש למסור לשאר הפונקציות של חבילה זו.

הפונקציות עצמן והקוד שלהן תוגדרנה בקובץ אחר שמכונה בדרך כלל **קובץ המקור** (source file). קובץ זה מוסתר מהמשתמש, ולכן למתכנת המשתמש בחבילת התוכנה יש ממשק קשיח שרק בו הוא יכול להשתמש. נגדיר צומת ברשימה אשר לו מצביע **לצומת הבא** (next) ומצביע **לנתון** (data) שנמצא בצומת הנוכחי.

```
typedef struct node_t {
      struct node *next;
      void *data;
} node;
```

הרשימה עצמה היא רשומה שמכילה מצביע לצומת (node, התא) הראשון, ומצביע לצומת הנוכחי. המצביע האחרון (current) משמש למעבר על הרשימה. מצביע זה מצביע על הצומת הנוכחי בכל שלב של האיטרציה.

```
/* the structure of the list */
struct list_t {
      node *head;
      node *current;
      int nitems;
} list;
```

הפונקציה create יוצרת רשימה חדשה ומאתחלת אותה. מספר האלמנטים ברשימה והמצביעים מאותחלים לאפס. הפונקציה מחזירה מצביע לאזור זיכרון אשר הוקצה בעזרת פונקציית הספרייה malloc.

```
/* creates a new list and returns a pointer to it */
list *list_create()
{
      list *new_list = (list*)malloc(sizeof(list));
      new_list->head = new_list->current = NULL;
      new_list->nitems = 0;
      return new_list;
}
```

הפונקציה list_add יוצרת צומת חדש ומחברת אותה לרשימה. המצביע לנתונים בצומת החדש מאותחל למצביע (data) הנתון לפונקציה כארגומנט. כלומר, המצביע data בצומת החדש יצביע לנתון החדש, ואילו המצביע next יצביע לראש הרשימה הקודם. לאחר מכן, המצביע לראש הרשימה מעודכן לתא החדש. לבסוף, הפונקציה מגדילה את מספר האלמנטים ברשימה.

```
void list_add(list *listptr, void *data)
{
      node *nnode= (node*)malloc(sizeof(node));
      nnode->next = listptr->head;
      nnode->data = data;
      listptr->head = nnode;
      listptr->nitems++;
}
```

הפונקציה list_first מאתחלת את האיטרציה (מעבר) על הרשימה ומחזירה מצביע לאלמנט הראשון ברשימה. אם הרשימה ריקה, מוחזר מצביע מאופס המסמן תנאי זה.

```
void *list_first(list *lptr)
{
      lptr->current = lptr->head;
      return (lptr->current ? lptr->current->data : 0);
}
```

הפונקציה list_next מקדמת את המצביע לצומת הנוכחי ומחזירה מצביע אל הנתונים שאליהם מתייחס צומת זה. אם אין יותר רשומות ברשימה מוחזר מצביע מאופס.

```
void *list_next(list *lptr)
{
      if (lptr->current != NULL)
            lptr->current = lptr->current->next;
      return (lptr->current ? lptr->current->data : 0);
}
```

כאמור, כל הפונקציות המטפלות ברשימה מופיעות בקובץ שהמשתמש במודול **אינו** חשוף לו. קובץ זה מהדרים בנפרד והמתכנת משתמש בתוצאת ההידור. לכן, כל שינוי בפונקציה כזו, לא יגרום לשינויים בתוכנת המשתמש. יתרה מכך, יש לנו עכשיו מודול שמספק לנו שירותים בסיסיים של רשימה, ואנו יכולים להשתמש בו לרשימות רבות.

למודול עדיין חסרה פונקציה שתשחרר את הזיכרון שהוקצה במהלך הוספת אלמנטים לרשימה.

כתרגיל, אני משאיר לך ליצור פונקציה זו.

## 2.2.4 החסרונות של המודולים

המודולים אכן פתרו בעיה חשובה. כעת הוסרה התלות בין מספר גדול של פונקציות ומבנה נתונים ומשתנים רבים. כעת יש אפשרות לשנות מבנה נתונים ביתר קלות מבלי ששאר חלקי התוכנה ישתנו. במילים אחרות: כך הושגה **הסתרת המידע** (information hiding). כלומר, פונקציות שאינן שייכות למודול, אינן מכירות את מבנה הנתונים של המודול. תוכנית הנוקטת בגישה זו המורכבת מאוסף של מודולים, כאשר כל אחד מהם מספק כממשק רשימה של פעולות, או פונקציות שירות אשר בהן הוא תומך - נקראת **תוכנית מודולרית** (modular program).

למרות היתרון שהושג בעזרת המודולים, עדיין קיימים מספר חסרונות. כדי להבין אותם, נראה כיצד להשתמש במודול כזה:

```
int main()
{
    struct list *l1, *l2;
    l1 = list_create();
    l2 = list_create();
    /*  adding some elemnts to the list */
    list_add(l1, elm1);
    list_add(l2, elm2);
    /* What to do here ? How do we destroy elements in the
      list*/
    ...
    list_destroy(l1);
}
```

עצם ההכרזה על המשתנים l1 ו-l2 אינה הגדרה של הרשימות, ואינה יוצרת רשימות, או מקצה להם זיכרון במחשב. לכן, אם המשתמש אינו קורא לפונקציות שיוצרות את הרשימה וקורא במקומן לפונקציה, כגון list_add (שנועדה להוספת נתון לרשימה), תופסק התוכנית בנקודה זו, במקרה הטוב. במקרה הגרוע, תמשיך התוכנית לרוץ, אך תוצאותיה אינן צפויות ובוודאי שלא כפי ש"התכנון המשורר". במקרה זה יתרחשו בעיות שונות, שלכאורה נראות ללא שום קשר לבעיית האתחול. לכן, על המתכנת לזכור שעליו ליצור רשימה חדשה על ידי קריאה לפונקציה שיוצרת ידית לרשימה זו. המתכנת הגדיר ידית כזו, אך היא אינה מספיקה.

המשתמש במודול הרשימות צריך לעבור על הרשימה ולשחרר את הזיכרון שהקצה. בנוסף, עליו לקרוא לפונקציה שהורסת את הרשימה. זהו תהליך שטומן בחובו אפשרות לשגיאות רבות. למשל, מתכנת המשתמש ברשימה עלול לשכוח לשחרר אותה.

חיסרון אחר של המודול הזה הוא הסרבול בשימוש בו. כאשר קוראים לפונקציה, יש להעביר את המצביע לרשימה הנוכחית, כלומר, את הידית שיצרנו בשלב מוקדם יותר בעזרת הפונקציה create_list.

לאור החסרונות האלה, הגיעו מפתחים ומהנדסים רבים למושג מחלקה - class.

## 2.3 המחלקה

**מחלקה** (class) היא מבנה נתונים ואוסף של פעולות שמוגדרות על מבנה הנתונים. אפשר להגדיר משתנים מסוג מחלקה. משתנים אלה הם מופעים של המחלקה, או אובייקטים. הציון **אובייקט** (object) חשוב במיוחד, ויש להבדיל בין מחלקה לאובייקט. מחלקה היא תבנית, כמו רשומה, המאפשרת להגדיר אובייקטים, כלומר, משתנים השייכים למחלקה.

הפעולות (פונקציות) קשורות למבנה הנתונים של המחלקה ואינן עומדות בזכות עצמן. פעולות אלו נקראות **שיטות** - **methods**, או **פונקציות חברות** - **member-functions**.

כשמגדירים אובייקט של מחלקה מסוימת, באופן אוטומטי מתבצע האתחול שלו. כשיוצאים מהבלוק שבו הוגדר האובייקט, האובייקט מפורק באופן אוטומטי. זאת בניגוד לדוגמה הקודמת של המודול, בו המתכנת היה צריך לזכור ליצור ולפרק משתנים (אובייקטים) של המודול.

לפיכך, המחלקה פותרת את החסרונות שראינו במודול. השימוש במחלקה פשוט וקל יותר. עובדה זו חשובה מאוד. אם נחזור לרשימה שלנו, הרי שהזמן אותו נשקיע בפיתוח הרשימה קצר יחסית. לעומת זאת, נשתמש ברשימה בהרבה מקומות, ולכן נוחיות השימוש בה חשובה מאוד.

כיצד, אם כן, נראית הצהרה על מחלקה ב-C++? הצהרה של מחלקה מתחילה במילת המפתח **class**. בהמשך ההצהרה מופיעים **החברים** (members) במחלקה. חברים אלו יכולים להיות **שדות נתונים** (fields), או **פונקציות** (functions) למשל:

```
class list {
      node *head;
      node *current;
      int nitems;
public:
      list();
      ~list();
      void add(void *);
      void *first();
      void *next();
};
```

החלק הראשון של המחלקה, עד למילת המפתח **public** (**ציבורי**) זהה להגדרת המבנה list ב-C. השינוי היחיד הוא, שבמקרה זה אנו משתמשים במילת המפתח class. מילת המפתח public מציינת את האזור של המחלקה שפתוח לשימוש על ידי מחלקות, או פונקציות אחרות. כלומר, רמת הגישה לאזור זה של המחלקה היא ציבורית.

C++ היא שפת כלאיים, אפשר להגדיר בה **פונקציות גלובליות** (global functions), שאינן שייכות למחלקה, או מחלקות ואובייקטים. שפות תכנות מונחות אובייקטים טהורות אינן מאפשרות הגדרה של פונקציות גלובליות. יש הרואים דבר זה כחיסרון ויש הרואים זאת כיתרון.

המצדדים בשפות תכנות מכוונות אובייקטים טהורות, טוענים שהאפשרות להגדיר פונקציות גלובליות מאפשרת למתכנת "לקלקל" את מבנה התוכנה. המצדדים בשפות כלאיים טוענים שיש בשפה כמו C++ יכולת גדולה יותר, והשפה מאפשרת ליהנות משני העולמות בעת ובעונה אחת. אני נוטה להסכים עם הקבוצה השנייה.

בדוגמה שלעיל מוגדרות חמש פונקציות. הראשונה מביניהן היא בעלת שם זהה לזה של המחלקה. זוהי פונקציה מיוחדת שמשמשת לאתחול, או ליצירת אובייקטים חדשים של המחלקה. לפונקציה שם מיוחד **constructor** או **בנאי**. פונקציה זו נקראת באופן אוטומטי על ידי המהדר, ואין באפשרות המתכנת לקרוא לה באופן אחר. הפונקציה נקראת כאשר מוגדר אובייקט של המחלקה.

הפונקציה השנייה היא **המפרק**, או **destructor**. פונקציה זו נקראת באופן אוטומטי על ידי המהדר כשהמשתנה של המחלקה (**אובייקט**) יוצא מתחום ההגדרה. כלומר, המהדר מבצע את פעולות ההריסה של אובייקטים באופן אוטומטי.

שאר הפונקציות: add, first, next הן פונקציות רגילות של המחלקה, אשר מבצעות את הפעולה של הוספת צומת לרשימה ומעבר על הרשימה. השימוש באובייקטים של המחלקה נראה באופן הבא:

```
int main()
{
	list l1, l2;
	void *ptr;
	l1.add("some data");
	l2.add("another item");
	for (ptr = l1.first(); ptr; ptr = l1.next())
		printf("%s\n", ptr);
	return 0;
}
```

התוכנית הקטנה הזו מגדירה שתי רשימות (אובייקטים) l1 ו-l2. התוכנית קוראת לפונקציה add פעמיים, עבור כל אחד מהאובייקטים l1 ו-l2. הקריאה לפונקציה של אובייקט נעשית על ידי כתיבת שם האובייקט, לאחריו נקודה ולאחריו שם הפונקציה. דבר זה דומה לגישה לשדה בשפת C. מעבר על הרשימה והדפסת המחרוזות הנמצאות ברשימה נעשה באופן דומה.

דוגמה זו שונה מהדוגמה הקודמת, שבה השתמשנו במודולים. כזכור, היינו צריכים לקרוא לפונקציה שיצרה ידית לרשימה. בנוסף, היינו צריכים להרוס את הרשימה במפורש. ב-++C כל הפעולות האלו מתבצעות באופן אוטומטי על ידי המהדר. הדבר חוסך את הטירחה בכתיבת פעולות אתחול והריסה של מודולים, וגם מונע מאיתנו שגיאות טיפשיות, כמו לנסות להשתמש בידית (אובייקט) שלא יצרנו כלל.

דבר נוסף שאפשר לראות מקטע התוכנה הקודם הוא האלגנטיות הרבה. אין צורך להעביר מצביע לאובייקט בכל קריאה לפונקציית שירות של המחלקה. האובייקט הנוכחי הוא הידית של פונקציית השירות.

**פונקציות חברות** (member functions) במחלקה X כלשהי, הן פונקציות שנכתבות בצורה הבאה:

```
void X::func(arg-list)
{
      statments...
}
```

שם המחלקה מופיע לפני שם הפונקציה. לאחר שם המחלקה מופיעות פעמיים **נקודתיים**. סימון זה מציין שהפונקציה **שייכת** למחלקה מסוימת, ואינה פונקציה גלובלית (שאינה שייכת לאף מחלקה). לאחר שם הפונקציה מופיעה רשימת הארגומנטים המועברים לפונקציה, כמו ב-C. רשימת הארגומנטים נמצאים בין סוגריים. גוף הפונקציה נמצא בין סוגריים מסולסלים בדומה ל-C.

כדי להדגים זאת, נראה את הפונקציה add של הרשימה, שמוסיפה אלמנט לרשימה. פונקציה זו מקבלת מצביע לאלמנט שיש להוסיפו לרשימה. הפונקציה משתמשת בפונקציית המערכת new של ++C. new מקצה זיכרון, לפי גודל האובייקט שנמסר לה, ומחזירה מצביע לזיכרון זה. אין צורך להמיר מצביע זה לסוג של המצביע הנוכחי כמו במקרה של malloc ב-C.

שינוי נוסף לעומת C הוא שאין צורך להעביר את המצביע לרשימה הנוכחית, ואין צורך להשתמש במצביע זה כדי לגשת לשדות של הרשימה.

```
void list::add(void *data)
{
      node = new node;
      node->data = data;
      node->next = head;
      head = node;
      nitems++;
}
```

ב-++C יש ארגומט נוסף אשר מסופק על ידי המהדר לכל פונקציה של מחלקה. ארגומנט זה הוא **this**. המתכנת אינו צריך לספק ארגומנט בקריאה לפונקציה, אך לעומתו יכול כותב הפונקציה להשתמש בארגומנט זה כמצביע לאובייקט הנוכחי. בהמשך נראה את השימוש של המצביע.

כל פעולה על שדה של המחלקה מתבצע יחסית למצביע לאובייקט הנוכחי this. כלומר, השורה:

```
head = node;
```

שקולה לשורה הבאה:

```
this->head = node;
```

פונקציה מיוחדת של המחלקה היא ה**בנאי** (constructor). כאמור, הפונקציה נקראת כאשר מגדירים אובייקטים מהמחלקה הנתונה. הבנאי למחלקה X כלשהי מוגדר בצורה הבאה:

```
X::X(arg-list)
{

}
```

שם הפונקציה במקרה זה, הוא כשמה של המחלקה. רשימת הארגומנטים יכולה גם להישאר ריקה. למשל, במקרה של רשימה, אפשר לכתוב בנאי עם רשימת ארגומנטים ריקה:

```
list::list()
{
      head = NULL;
      current = NULL;
      nitems = 0;
}
```

הבנאי מאתחל את שדות הפרטים של הרשימה כדי לאפשר פעולה תקינה שלה. מומלץ לכתוב בנאי קצר שמעביר את האובייקט למצב תקין ומוכן לפעולה. עדיף להימנע מבנאים ארוכים שמבצעים אלגוריתמים שלמים בבנאי עצמו.

פונקציה מיוחדת אחרת היא פונקציית מחיקה וניקוי, **המפרק** (destructor) של המחלקה. פונקציה זו נכתבת על ידי מתכנן המחלקה ותפקידה למחוק ולנקות תוצאות של פעולות ביניים שנעשו על אובייקט של המחלקה. למשל, לשחרר הקצאות זיכרון שנעשו על ידו.

הצורה הכללית של פונקציה כזו עבור מחלקה כלשהי X היא כזו:

```
X::~X()
{
      statements...
}
```

התחביר של המפרק דומה לתחביר של הבנאי, אך נמצא כאן שני שינויים. השינוי הראשון הוא: שם הפונקציה הוא כשם המחלקה, אך בתוספת הסימן "~" לפני שם הפונקציה. השינוי השני הוא, שהמפרק אינו מקבל ארגומנטים.

לשתי פונקציות אלו (הבנאי והמפרק) אין אפשרות להחזיר ערך כלשהו. אין אפשרות לראות אם הבנאי נכשל בפעולתו ולכן יש לדאוג לכך שהוא לא ייכשל. דוגמאות לכך נראה בהמשך. במקרה של הרשימה צריך המפרק לשחרר את כל הצמתים של הרשימה; המפרק יעבור על כל צומת וישחרר אותו.

```
list:~list()
{
      node *cur = head, *next;
      while (cur != NULL) {
            next = cur->next;
            delete cur;
            cur = next;
      }
}
```

במפרק אנו מגלים מילת מפתח חדשה: **delete** (**מחיקה**, **ביטול**). למעשה, זוהי קריאה לפונקציה של המערכת שמשחררת זיכרון, שהוקצה קודם לכן בעזרת הפונקציה **new**.

**חוק:** אין להפעיל את delete על זיכרון שלא נרכש בעזרת new. הפעלה של delete על זיכרון שלא נרכש בעזרת new יגרום לבעיות ולתופעות בלתי צפויות.

## 2.4 סיכום

בפרק זה ראינו את הגורמים ליצירת מודולים. ליישום (application) שמפתחים יש מחזור חיים ארוך, שבמהלכו נוספים לו דרישות שגוררות שינויים מרחיקי לכת בתוכנה, עד כדי שיכתובה מחדש.

כדי להתגבר על בעיות אלו הומצא המודול. מודול מספק שירותים **ומסתיר** את אופן המימוש משאר חלקי התוכנה. לכן, כאשר משתנה אופן המימוש, ישתנה רק המודול ולא כל התוכנה המשתמשת בו. לפיכך, משיג המודול מידור של רכיבי התוכנה.

יש מספר חסרונות במודולים. למשל, יש צורך שהמתכנת המשתמש בהם יאתחל ויהרוס אותם. אם המשתמש שוכח ליצור אובייקט מתאים של המודול, אין דבר המונע ממנו למסור ידית כזו לפונקציות השירות של המודול ולגרום לתוצאות בלתי צפויות.

כדי לפתור חסרונות אלה מגדירים את המחלקה. משתנים, או מופעים, של מחלקה נקראים **אובייקטים**. כאשר מגדירים אובייקט כזה, נקראת פונקציית האתחול שלו, ה**בנאי**, באופן אוטומטי. כאשר יוצאים מהבלוק אשר בו הוגדר האובייקט, פונקציית הניקוי, **המפרק**, נקראת באופן אוטומטי גם כן כדי לבטל הקצאות והקצאות שונות. בהמשך נראה איך לכתוב פונקציות אלו.

## 2.5 תרגילים

1. הגדר את הפונקציה של המודול list_destroy. לאיזו פונקציה של המחלקה list שקולה פונקציה זו?
2. הגדר מודול למערך דינמי, המגדיל את המערך כאשר אין בו מספיק מקום. הגדר פונקציות בנייה ופירוק למערך זה.
3. כתוב תוכנית המנהלת רשימת טלפונים בעזרת המערך שבנית.
4. האם המפרק של הרשימה צריך לשחרר את הזיכרון אליו מצביעים הצמתים בעזרת data? נמק!
5. כתוב את הפונקציות המאפשרות מעבר על הרשימה first ו-next.
6. כתוב פונקציה המבטלת צומת ברשימה. מה הפונקציה צריכה להחזיר למשתמש?

# פרק 3

# המחלקה

בפרק זה נמשיך ללמוד את המחלקה ב-C++, ונרחיב את ההקדמה מהפרק הקודם.

המחלקה היא אבן הבניין הבסיסי בתכנות מוכוון אובייקטים, והיא מחליפה את המודול ומספקת שירותים והסתרת מידע. מטרתנו בפרק זה היא ללמוד את הפונקציות השונות של מחלקה הכוללות **אופרטורים** שונים, **פונקציות בנייה ופונקציות פירוק** (ובקצרה - **בנאים ומפרקים**). בסוף פרק זה נדע לכתוב מחלקות שימושיות למספר יישומים.

בפרק זה נתייחס למחלקות ואובייקטים. מחלקות הן **תבניות** (templates) המורות למהדר כיצד להגדיר אובייקטים. אובייקטים הם משתנים מסוג של מחלקה נתונה. **אובייקט** (object) הוא מופע מסוים של מחלקה. בזמן ריצה חיים האובייקטים, ואילו בזמן הידור המחלקות משמשות כ**תבנית חוקים** למהדר, כדי להגדיר את הפעילות שיכולה להתבצע על אובייקטים מסוג מסוים. בנוסף, מגדירה התבנית את השדות של האובייקטים ששייכים למחלקה מסוימת.

## 3.1 מבנה המחלקה - אזורי גישה

למחלקה יש אזורי גישה שונים. החלוקה לאזורים נועדה לשמור את הסתרת המידע ממשתמשי המחלקה.

מילת המפתח public מציינת שיש גישה לחלק זה של המחלקה, לפונקציות, או לאובייקטים אחרים שלא שייכים למחלקה הנוכחית. החלק של המחלקה שאינו נמצא בחלק הציבורי, נחשב חלק פרטי וזוהי ברירת המחדל. לחלק הפרטי של המחלקה יכולות לגשת רק פונקציות השייכות למחלקה.

במחלקה כלשהי X אפשר להגדיר את אזורי הגישה למחלקה בצורה הבאה:

```
class X {
      int xval;
public:
      int yval;
```

```
private:
      int zval;
};
```

מילת המפתח **private** (פרטי) מציינת את החלק הפרטי של המחלקה, אשר אליו יכולות לגשת רק פונקציות החברות במחלקה. מילות מפתח אלו יכולות להופיע מספר פעמים בהגדרה של המחלקה, כאשר כל הופעה משנה את הגישה לשדות אשר מופיעים אחריה. ההגדרה תקפה עד להגדרה הבאה.

במחלקה שהגדרנו כאן, המשתנה xval נמצא בחלק הפרטי של המחלקה, ולכן הגישה אליו מוגבלת. לעומת זאת, השדה yval נמצא בחלקה הציבורי של המחלקה ולכן הגישה אליו אפשרית לכל הפונקציות והאובייקטים. השדה zval נמצא בחלק הפרטי ולכן הגישה אליו מוגבלת גם היא.

נחזור למושגים אלה בהמשך פרק זה.

## 3.2 הבנאי - פונקציית האתחול של המחלקה

בפרק הקודם ראינו שהמהדר משתמש **בפונקציית בנייה** (constructor), או **בנאי**, כדי לאתחל אובייקטים שונים. לכן, אחת הפונקציות החשובות ביותר של המחלקה היא הבנאי. למעשה, הבנאי הוא הוראות למהדר כיצד לאתחל אובייקט חדש של המחלקה.

לכל מחלקה יכולים להיות מספר בנאים, שלהם תפקידים שונים. נסקור בסעיף זה את הבנאים השונים. כדי להדגים אותם נשתמש במחלקה מהפרק הקודם, ובעת הצורך נוסיף מחלקות נוספות.

### 3.2.1 בנאי ברירת המחדל

**בנאי ברירת המחדל (default constructor)** אינו מקבל פרמטרים כלשהם. כבר נתקלנו בבנאי זה בפרק הקודם כשדיברנו על המחלקה list.

```
list::list()
{
      head = 0;
      nitems = 0;
      current = 0;
}
```

בנאי זה מאפס את השדות של הרשימה, מביא אותה למצב כשיר לפעולה, ומאפשר להגדיר אובייקטים בצורה הבאה:

```
list lobj;
```

כלומר, אין צורך לספק פרמטרים לבנאי זה. הוא שימושי במיוחד כאשר רוצים להגדיר מערך של אובייקטים, או במקרה הזה מערך של רשימות:

```
list listArray[5];
```

השורה הקודמת מגדירה מערך של חמש רשימות. לאתחול כל רשימה משתמש המהדר בבנאי ברירת המחדל, לכן נקבל חמש רשימות ללא אף צומת, כאשר מספר האובייקטים בכל רשימה הוא אפס.

אין אפשרות להגדיר מערך של אובייקטים, אם למחלקה של האובייקטים אין בנאי ברירת המחדל, מבלי לציין את הערך ההתחלתי של כל אובייקט; כלומר, את הארגומנטים של אתחול עבור כל אובייקט. יש לנו, למשל, את המחלקה הבאה:

```
class X {
      int x;
public:
      X(int xo);
      ...
};
```

מכיון שאין בנאי ברירת מחדל למחלקה X, יש לאתחל כל אובייקט באופן נפרד כך:

```
X xarr[] = { X(0), X(1), X(2) };
```

## 3.2.2 בנאי עם פרמטרים

בנאים המקבלים פרמטרים משתמשים בהם כדי לאתחל אובייקט חדש. נניח למשל, שיש לנו מחלקה המשמשת כמחרוזת של C.

```
class String {
      char *str;
public:
      String(const char *);
      ~String();
};
```

הבנאי של המחלקה String מקבל מצביע לתו. מילת המפתח **const** (קבוע) מסמנת שהבנאי לא ישנה את תכולת המצביע שמועבר אליו. כאשר מילת מפתח זו מופיעה לפני משתנה, היא מודיעה למהדר שיש לשמור על המשתנה, כדי שהמתכנת לא ישנה את תכולתו בטעות. נחזור למילת מפתח זו מאוחר יותר. הבנאי מעתיק את המחרוזת שניתנה לו למקום חדש בזיכרון אותו מקצה הבנאי:

```
String::String(const char *s)
{
      str = new char[strlen[s) + 1];
      strcpy(str, s);
}
```

נשים לב, שבמקרה זה השימוש באופרטור **new** מלווה בסוגריים מרובעים שמציינות הקצאת זיכרון של מערך. הקצאה של מערך מחייבת שימוש באופרטור **delete** של מערך. שימוש באופרטור אחר יגרום לתוצאות בלתי צפויות. לכן, במפרק של מחלקה זו מופיע משפט delete באופן הבא:

```
String::~String()
{
      delete [] str;
}
```

בנאי עם פרמטר אחד יכול לשמש כאופרטור המרה מסוג הפרמטר לאובייקט של המחלקה. נחזור לזה בהמשך. **בנאי מחלקה הוא פונקציה לכל דבר**, פרט לעובדה שהוא נקרא באופן אוטומטי על ידי המהדר והמתכנת אינו יכול לקרוא לו.

כמו כל פונקציה אחרת, הבנאי יכול לקבל מספר פרמטרים. למשל, הבנאי יכול לקבל מצביעים לשתי מחרוזות ולשרשר ביניהם באופן הבא:

```
String::String(const char *s1, const char *s2)
{
      int r1 = strlen(s1), r2 = strlen(s2);
      str = new char[r1 + r2 + 1];
      strcpy(str, s1);
      strcpy(s1+r1, s2);
}
```

הבנאי הזה מחשב את אורכי שתי המחרוזות, מקצה זיכרון באורך של שתי המחרוזות ותו נוסף. התו הנוסף נועד לתו המיוחד בעל ערך 0 (ערך דצימלי 0), שמסמן את סוף המחרוזת. הערך של התו המיוחד הוא 0 (שלם) וחשוב לא להתבלבל בינו לבין הערך '0' של ASCII שאינו ערך אפס (שלם). לאחר מכן, מעתיק הבנאי את שתי המחרוזות, זו לאחר זו, לתוך אזור הזיכרון החדש שאליו מתייחס המצביע str. אופן השימוש בבנאי זה הוא:

```
String s2("str one", " and its tail");
```

האובייקט s2 הוא אובייקט מסוג String אשר "מכיל" מצביע למחרוזת המשורשרת משתי המחרוזות אשר הועברו לבנאי. בנקודת ההגדרה של אובייקט זה נקרא הבנאי בעל שני הפרמטרים באופן אוטומטי, כדי ליצור את האובייקט.

### 3.2.2.1 בנאי עם קבועים

ראינו שניתן להגדיר משתנים קבועים. כאשר ההגדרה של משתנה כוללת את מילת המפתח **const**, המשמעות היא שאין לשנות את תכולת המשתנה. המהדר מגן על משתנה כזה ודוחה את האפשרות לשנות את תכולתו. למשל, המשתנים הבאים הם קבועים:

```
const int ok = 1;
const char dot = '.';
const float pi = 3.14;
const char *str = "A constant string";
```

מכיון ששלושת המשתנים הראשונים הם קבועים (כלומר, const), הם אינם יכולים להופיע בצד השמאלי של משפט ההשמה. לעומת זאת, המשתנה השמאלי הוא מצביע לתווים קבועים. כלומר, אין אפשרות לשנות את התכולה של המצביע:

```
*str = 'a';              // not allowed
str = "Another string";
```

המשפט הראשון בדוגמה זו אינו חוקי, כי הוא מנסה לשנות את תכולת המצביע. המשפט השני הוא חוקי משום שאין משנים את התכולה של המצביע, אלא את המקום שאליו המצביע מתייחס (זוהי המשמעות של מצביע). אם ברצוננו להגדיר מצביע קבוע, שמצביע למקום אחד בלבד, עלינו לעשות זאת בצורה הבאה:

```
char *const cstr = "Str";
*cstr = 's';                      // Ok this allowed
cstr = "Another string";
```

ההשמה הראשונה חוקית, ולכן היא מאפשרת לשנות את אזור הזיכרון שאליו המצביע מתייחס. לעומת זאת, ההשמה השנייה אינה חוקית, משום שאין לשנות מצביע קבוע כדי שיצביע למקום אחר.

לפיכך, משתנה קבוע יכול להיות מאותחל פעם אחת ויחידה בעת הגדרתו.

## 3.2.3 בנאי העתקה ומושג הייחוס

לכל מחלקה כלשהי X מוענק באופן אוטומטי **בנאי העתקה**. כלומר, גם אם אתה, המתכנת, אינך מגדיר בנאי כזה, המהדר ייצר אותו עבורך. מהו אם כן בנאי ההעתקה?

עבור מחלקה נתונה X, בנאי ההעתקה מוגדר כך:

```
X::X(const X &xref)
{
        ...
}
```

בהגדרה זו אנו נתקלים במושג חדש שנקרא **ייחוס** (reference), או הפנייה לאובייקט. פעולות על הייחוס ישפיעו על האובייקט אשר אליו מתייחס הייחוס. כלומר, הפרמטר של הבנאי הזה הוא שם לאובייקט אחר, ובדומה למצביע, קריאה לבנאי זה אינה מעבירה את האובייקט כולו אל המחסנית, אלא את שמו בלבד. כלומר, נחסכת ההעתקה של האובייקט אל המחסנית. נחזור בהמשך למושג זה.

כאשר המהדר מעניק בנאי העתקה למחלקה מסוימת, הבנאי מבצע העתקה של החברים במחלקה מאובייקט אחד לאחר. הבנאי שהמהדר מעניק כברירת המחדל מבצע **העתקה רדודה** (shallow copy), של שם בלבד . למשל, הבנאי נקרא לבצע משפטים כמו אלה:

```
String s1("kookoo");
String s2(s1);
String s3 = s1;
```

בשורה הראשונה מוגדר אובייקט s1, בשורה השנייה מועתק האובייקט על ידי בנאי העתקה לאובייקט החדש s2. בשורה השלישית מועתק s1 לאובייקט החדש s3 על ידי בנאי העתקה. נזכור שבנאי העתקה, המוענק על ידי המהדר, **מעתיק את שמות השדות ולא את תוכנם**. לכן, לאובייקטים s2 ו-s3 יש כעת מצביעים (מסוג str) לאותו אזור בזיכרון בו נמצא האובייקט s1. פונקציית הפירוק של האובייקטים האלה תשחרר את אזור הזיכרון שלוש פעמים. מצב זה יכול להוביל לתוצאות בלתי צפויות של התוכנית, שבמקרה הטוב נזהה אותן במקום הפירוק של אובייקטים אלה. במקרה הרע, תהיינה התוצאות במקומות ללא קשר לכאורה לאובייקטים שאנו עוסקים בהן.

**חוק:** כאשר יש מצביעים במחלקה, עלינו להגדיר **בנאי העתקה**. הדבר נכון במיוחד למחלקות שאסור שתהיה להן אפשרות של העתקה רדודה (של שמות) בלבד.

בנאי ההעתקה אשר נכתוב למקרה זה, ייראה כך:

```
String::String(const String &s)
{
      str = new char[strlen(s.str) + 1];
      strcpy(str, s.str);
}
```

בנאי ההעתקה, במקרה זה, מקצה אזור זיכרון חדש ומעתיק לתוכו את המחרוזות של האובייקט המועבר כפרמטר לבנאי. לפיכך, ישחררו המפרקים אזורי זיכרון שונים והבעיה הקודמת נפתרת.

בנאים של אובייקטים נקראים בשני מקרים:

- כאשר מגדירים אובייקט במקום כלשהו של התוכנית.
- כאשר מקצים אובייקט בעזרת אופרטור new.

במקרה השני מוקצה לאופרטור new זיכרון כגודלו של האובייקט הנתון, ואז נקרא הבנאי המתאים לאובייקט. הבנאי המתאים הוא זה המקבל רשימת ארגומנטים מתאימה לאלה שנמצאים בנקודת היצירה של האובייקט. למשל:

```
String *stringPtr = new String("heap string");
```

במקרה זה, האובייקט מוקצה בזיכרון החופשי של התוכנית, ולכן גם האובייקט והמצביע str מוקצים לזיכרון זה.

## 3.3 המפרק, פונקציית הניקוי של אובייקטים

**פונקציית פירוק** (destructor), או **מפרק**, היא פונקציית הניקוי אשר מנקה הקצאות, הגדרות ומשאבים לאחר "מות" האובייקט. אלגוריתם הניקוי שמופעל הוא באחריות המתכנת שכתב את המחלקה שלה שייך האובייקט. המהדר מצידו, קורא לפונקציית הפירוק באופן אוטומטי כשהאובייקט יוצא מתחום הגדרתו, או כאשר מופעל על הכתובת שלו האופרטור **delete**. בסעיף זה נראה דוגמאות והסבר מפורט לכך.

סגנון כתיבה טוב הוא זה שמנקה ומשחרר את כל המשאבים שנתפסו על ידי האובייקט במהלך חייו. משאבים אלה יכולים להיות זיכרון של המחשב, או **מתארי הקבצים** (file descriptors) שמתקבלים כתוצאה משימוש בפונקציית המערכת open.

המפרק של מחלקה נתונה X, נראה כך:

```
X::~X()
{
      // ...
}
```

כמו כל פונקציה של מחלקה, מוגדר המפרק כשלפניו מופיע שם המחלקה. שם המפרק הוא כשם המחלקה כאשר קודם לו התו "~". שני תווי לוכסן "//" המופיעים בגוף המפרק הם סימנים להערות. **הערה** (comment) כזו מתחילה מצמד התווים "//" וממשיכה עד לסוף אותה השורה.

במחלקה שתוארה קודם לכן (string) ייראה המפרק כך:

```
String::~String()
{
      delete [] str;
}
```

מפרקים של אובייקטים נקראים אוטומטית על ידי המהדר כאשר אובייקט יוצא מהתחום בו הוגדר. המפרק נקרא כדי להרוס את האובייקט כאשר הבקרה של התוכנית עוזבת את הבלוק בו הוא הוגדר. הדבר קורה מייד לאחר שבוצע המשפט האחרון בבלוק בו מוגדר האובייקט. למשל:

```
void f(char *str)
{
   String s(str);
   // ... body of function
   // here just before the end of function dstor gets called
}
```

הקריאה למפרק של אובייקט s מתבצעת ממש לפני היציאה מהפונקציה. עובדה זו נותנת לאובייקט אפשרות לנקות אחריו, ולהחזיר למערכת משאבים שהאובייקט נזקק להם במהלך חייו. במקרה זה הקצה האובייקט זיכרון דינמי ועליו מוטלת האחריות לשחרורו. משאבי מערכת אחרים יכולים להיות קבצים, או זיכרון משותף בין תהליכים. בכל מקרה כזה אחראי האובייקט לשחרור המשאבים בעת הפירוק. מועד הפירוק נקבע על ידי המהדר.

ייתכן שהאובייקט הוגדר בבלוק בקרה כלשהו בתוך בלוק בקרה אחר. במקרה כזה ייהרס האובייקט כאשר הבקרה של התוכנית יוצאת מבלוק הבקרה הפנימי. למשל:

```
void f1(char *str)
{
      if (some condition) {
            String s1(str);
```

```
        // ...
    }
}
```

בדוגמה זו תיקרא **פונקציית הפירוק** (destructor) על ידי המהדר כאשר מגיעים לסוף משפט if, כלומר הרבה לפני סוף הפונקציה f1.

פירוק האובייקט אינו נקרא באופן אוטומטי כאשר האובייקט מוקצה מתוך הזיכרון החופשי של התוכנית. כאשר אובייקט מוקצה בעזרת האופרטור **new**, פונקציית הפירוק של האובייקט אינה נקראת באופן אוטומטי. במקרים כאלה, האחריות להחזרת הזיכרון ופירוק האובייקט מוטלת על המתכנת. למשל:

```
void f3(const char *str)
{
      String *s1 = new String(str);
      //...
}
```

כשפונקציה זו מסתיימת, לא מתבצעת קריאה למפרק של האובייקט s1. מצב זה גורם **זליגת זיכרון** (memory leak). במקרה כזה יש להפעיל את האופרטור **delete** על האובייקט s1. הפעלת האופרטור delete תגרום קריאה אוטומטית למפרק (destructor) של האובייקט s1, וכתוצאה מכך, לשחרור הזיכרון שלהוקצה לו. הזיכרון שהוקצה **עבור** האובייקט (לא **על ידי** האובייקט) על ידי המתכנת בקריאה לאופרטור new, ישוחרר.

## 3.4 מושג הייחוס

נתקלנו כבר במושג הייחוס ובסעיף זה נבין מושג זה לעומקו. **הייחוס** (reference) הוא מושג חדש יחסית ל-C, אבל אינו קשור דווקא למחלקה ב-C++.

**הייחוס** הוא שם אחר לאובייקט. מתחילים רבים ב-C++ מתקשים להבין עובדה זו. נסביר זאת בעזרת דוגמאות.

משתנה מסוג ייחוס הוא שם אחר לאובייקט, אך הוא **אינו** אובייקט בפני עצמו. ב-C++ אפשר להגדיר ייחוס לכל סוג של משתנה (לאובייקט, או למשתנה בסיסי של השפה). המשמעות של ייחוס לאובייקט, או למשתנה בסיסי היא זהה, ודומה לסמנטיקה של מצביעים, למשל:

```
int i = 0;
int &iref (i);

i = 1;
printf("i=%d, iref=%d\n", i, iref);
```

ההדפסה של שני המשתנים בשורה לעיל תיתן אפס כמובן. המשתנה xref אינו משתנה בזכות עצמו, זהו שם אחר למשתנה i. הגדרה של משתנה מסוג ייחוס נעשית על ידי הוספת & לפני שם המשתנה.

משתנה מסוג ייחוס מאותחל פעם אחת ויחידה - בנקודת ההגדרה שלו. אי אפשר לשנות את הייחוס לאחר הגדרתו. כלומר, אי אפשר לקשור את הייחוס למשתנה אחר, לאחר הקשירה הראשונית בעת הגדרתו.

אם הייחוס מופיע מצידו השמאלי של משפט השמה, משתנה תכולת האובייקט שאליו מתייחס הייחוס ולא הייחוס עצמו (כמו האופרטור " * " על מצביע), כלומר:

```
iref = 1;
```

לאחר משפט ההשמה הזה יהיה גם ערכו של המשתנה i אחד. משתנה ייחוס דומה מבחינה זו למצביע רגיל של ++C. השוני בין מצביע לבין משתנה ייחוס הוא שאפשר לשנות מצביע, כלומר מצביע יכול להצביע לאובייקט אחר, אבל משתנה ייחוס מצביע מרגע הולדתו ועד מותו לאותו אובייקט. למשל:

```
int i1, i2;
int &iref(i1);
int *ip = &i1;

*ip = 1;
```

לאחר משפט השמה, הערך של i1 הוא 1. מצביע, לאחר אתחולו, יכול להצביע למשתנה אחר. למשל:

```
ip = &i2;
*ip = 2;
```

במקרה זה ערך המשתנה i2 ישתנה ל-2. אותה סדרה של פעולות, אך הפעם בעזרת ייחוס, תוביל לתוצאות שונות:

```
iref = 1;
iref = i2;
iref = 2;
```

בשורה הראשונה יקבל המשתנה i1 את הערך 1. בשורה השנייה הוא יקבל את הערך של המשתנה i2, ואילו בשורה השלישית הוא יקבל את הערך 2. שימו לב, בכל המקרים משתנה ערכו של המשתנה i1.

## 3.4.1 שימושי הייחוס

מתי, אם כן, נשתמש באפשרות הייחוס? **הייחוס** (reference) שימושי במיוחד כאשר רוצים להעביר אובייקטים גדולים לפונקציות. ב-++C הקריאה לפונקציה היא by value (לפי ערך). לפונקציה יש העתק של האובייקט שהוא הפרמטר, או הפרמטרים שלה. כאשר אובייקטים אלה גדולים, תוביל יצירת ההעתקה שלהם לביצועים ירודים של התוכנית. ראשית, התוכנית תצרוך יותר זיכרון, משום שיהיו לה מספר אובייקטים

במקום אחד. שנית, התוכנית תצרוך יותר זמן, מפני שיש צורך ליצור העתק של האובייקט. אולם, בשיטה זו פעולות שמשנות את האובייקט-פרמטר של הפונקציה, אינן משפיעות על האובייקט המקורי שנמסר לפונקציה. לדוגמה,

```
class String {
...
public:
...
      void set(const char *s);
      // set copies the new string onto str
};
```

נניח שיש לנו פונקציה חדשה בשם set למחלקה String. תפקיד פונקציה זו לשנות את המצביע הנתון לה למצביע חדש.

```
void f(String s)
{
      s.set("new value");
}

String s("old value");
f(s);
```

הקריאה לפונקציה f תגרום למהדר ליצור אובייקט זמני שהוא העתק של האובייקט המקורי s. הפונקציה תיקרא עם האובייקט הזמני, והשינויים שהיא תבצע על האובייקט הזמני לא ישפיעו על האובייקט המקורי.

החיסרון במקרה זה הוא שנוצר אובייקט זמני, שהוא העתק של האובייקט המקורי s, ונדרש זמן ריצה להעתקה וגם זיכרון נוסף. אם נגדיר את הפרמטר של הפונקציה f כייחוס, יעבור האובייקט בצורה הנקראת by reference. כך אין **שכפול** של האובייקטים והפונקציה פועלת על אובייקט המקור.

```
void f1(String &s)
{
      s.set("new value");
}

f1(s);
```

הפעלה של הפונקציה f1, במקרה זה, תגרום לשינוי של אובייקט המקור. בקריאה לפונקציה זו אין המהדר מייצר אובייקט זמני, לכן נשמרת היעילות. החיסרון שבשיטה זו הוא בכך שהפונקציה משפיעה על האובייקטים המקוריים. כדי לפתור בעיה זו אפשר להגדיר **ייחוס קבוע** (const reference), שאינו יכול להשתנות. ההגדרה של ייחוס קבוע נראית כך:

```
void f3(const String &s)
{
      s.set("new value");
}
```

מילת המפתח const המופיעה לפני המשתנה, מציינת את העובדה שאין אפשרות לשנות את ערך המשתנה שמועבר לפונקציה; כלומר, הפונקציה אינה משנה את הייחוס. במקרה כזה אסור לפונקציה להפעיל פונקציה אשר עלולה לשנות את האובייקט. **המהדר** (compiler) לא ירשה פעולה כזאת, ובזמן הידור נקבל הודעת שגיאה.

## 3.5 פונקציות מחלקה

ב-++C יש אפשרות להגדיר **פונקציות מחלקה** (member function) כלומר, פונקציות אשר שייכות למחלקה מסוימת. פונקציה כזו מכירה את מבנה הנתונים של המחלקה ופועלת על אובייקטים שלה. פונקציות כאלו דומות לפונקציות אחרות, אך שם המחלקה מתוסף לשם הפונקציה. לדוגמה, עבור מחלקה X ופונקציה כלשהי f של המחלקה, נרשום:

```
void X::f()
{
      // ... body of the function
}
```

לפני שם הפונקציה מופיע שם המחלקה, לאחריו "::" ולבסוף שם הפונקציה. פונקציית מחלקה יכולה לקבל פרמטרים בדיוק כמו פונקציה רגילה. אפשר להגדיר פונקציה המשנה את הערך של המחרוזת בצורה הבאה:

```
void String::set(const char *s)
{
      int len = strlen(s);
      delete [] str;
      str = new char[len + 1];
      strcpy(str, s);
}
```

הפרמטר של הפונקציה מוגדר כמצביע קבוע. כלומר, תוכן המצביע למחרוזת אינו יכול להשתנות על ידי הפונקציה set. הפונקציה משחררת את אזור הזיכרון הישן, מקצה אזור זיכרון חדש ומעתיקה לתוכו את המחרוזת הנתונה. פונקציה זו משנה את האובייקט עליו היא פועלת. הגישה לשדות באובייקט שבתוך פונקציה היא ישירה, ואין צורך להשתמש במצביע לאובייקט הנוכחי (this).

### 3.5.1 פונקציות קבועות

**פונקציה קבועה** (constant function) אינה משנה את האובייקט שעליו היא פועלת. הפונקציה הקודמת, set, אינה יכולה להיות פונקציה קבועה, מפני שהיא משנה את האובייקט שעליו היא פועלת. לעומתה, פונקציה שמחזירה את הערכים של האובייקט, או עושה חישוב כלשהו, יכולה להיות פונקציה קבועה.

פונקציה קבועה מאופיינת על ידי מילת המפתח **const**. עבור מחלקה X ופונקציה כלשהי קבועה f של המחלקה, מגדירים:

```
void X::f() const
{
      // ...
}
```

הפונקציה set של המחלקה String אינה יכולה להיות קבועה, כי היא משנה את האובייקט עליו היא פועלת. שתי הפונקציות הבאות הן קבועות:

```
class String {
//...
public:
//...
      const char *get() const;
      int length() const;
};

const char *String::get() const
{
      return str;
}

int String::length() const
{
      return strlen(str);
}
```

הפונקציות get ו-length אינן משנות את האובייקט שעליהן הן פועלות, ולכן הן יכולות להיות קבועות. פונקציות מסוג זה יכולות להיקרא עבור אובייקטים קבועים. למשל,

```
String s1("vary");
const String s2("const object");

s1.set("value");                  // ok
s2.set("value");                  // compilation error
int len = s2.length();            // ok
```

הקריאה לפונקציה set עבור האובייקט s2 תגרום לשגיאת הידור. המהדר יודע שהאובייקט האחרון הוא קבוע, ולכן אפשר להפעיל רק פונקציות קבועות עליו.

## 3.5.2 מנגנון inline

מנגנון **הסתרת המידע** (information hiding) יוצר מספר רב של פונקציות וקריאות לפונקציות. לעיתים, קריאה לפונקציה היא בזבוז זמן ריצה של המחשב. כל פנייה

לאובייקט נעשית דרך פונקציית שירות, ולכן עלול זמן הריצה של התוכנית לעלות באופן ניכר.

הפתרון לבעיה הזו הוא מנגנון **inline** (בתוך השורה), שבעזרתו נחסכת הקריאה לפונקציה, מכיון שקוד הפונקציה כלול בשורות הקוד של התוכנית. עבור מחלקה כלשהי X ופונקציה כלשהי f של המחלקה, הגדרת פונקציית inline נעשית כך:

```
class X {
      // ...
public:
      // ...
      void f() {//... }
};
```

כאשר המהדר רואה קריאה לפונקציה כזו, הוא מטפל בפרמטרים שלה כמו שהוא נוהג בפונקציה רגילה, אבל במקום לקרוא לפונקציה הוא משלב בקוד התוכנית את קוד הפונקציה. למשל:

```
class String {
      // ...
public:
      // ...
      const char *get() const { return str; }
};

String s("a string");
// ...
const char *str = s.get();
```

בנקודת הקריאה לפונקציה get המהדר מוודא שהפרמטרים והערך המוחזר מהפונקציה תואמים, ואז הוא מחליף יחליף את שם הפונקציה בקוד עצמו. במקרה זה המהדר מוודא שהמצביע למחרוזת הוא קבוע, ואז הוא מחליף את המשפט הנתון במשפט הבא:

```
str = s.str;
```

בכך תחסך הקריאה לפונקציה. פונקציה שהיא **inline** יכולה להיות מוגדרת מחוץ לגוף המחלקה בתוספת מילת המפתח **inline**, באופן הבא:

```
class String {
//...
      const char *get() const;
};
```

```
inline const char *String::get() const
{
      return str;
}
```

הגדרה זו שקולה להגדרה הקודמת ואין העדפות בין שתיהן. הגדרת פונקציה כ-inline אינה מבטיחה שהיא אכן תהיה כזו, כלומר אין חובה על המהדר להחליף את הקריאה לפונקציה בקוד שלה עצמו. הגדרה כזו היא בגדר המלצה למהדר. אם, למשל, יש משפט לולאה (מסוג for או while), בפונקציה כזו רבים הסיכויים שהפונקציה לא תהיה inline.

הגדרה של פונקציה כ-**inline** תגרום לשורות קוד רבות יותר, בדומה לפריסת מאקרו, ולכן יש להשתמש באפשרות זו בתבונה. אישית, איני ממליץ להגדיר פונקציות ארוכות, מעל שלוש שורות קוד, כפונקציות inline. במיוחד איני ממליץ להגדיר פונקציות inline כפונקציות המכילות מבני בקרה.

המושג **מאקרו** (macro) חדש לנו בהקשר לשפת C ולכן נסביר אותו בקצרה. מאקרו הוא קטע תוכנית דמוי-פונקציה, אשר מטופל ברמת קדם המהדר (pre-processor). מאקרו מתחיל בתו '#' ומילת המפתח (השמורה) '**define**', כך למשל: #define.

והינה מאקרו לדוגמה: #define x 5. מאקרו זה קובע שהתו x יוחלף בערך 5. על כן, בכל מקום שקדם-המהדר יזהה את התו x, הוא יחליף אותו בסיפרה 5, וכתוצאה - המהדר לא "יראה" תווי x, אלא ספרות 5 בלבד.

יש אפשרות להגדיר מאקרו בצורות שונות, כמו למשל,

```
#define round(x)  (lint)(x+0.5)
```

במקרה זה, קדם המהדר יחליף את round(x) בביטוי המתימטי העוקב. כאן לא תהיה כל קריאה לפונקציה, אלא הרצת פקודות רגילות, שהן יותר יעילות. עם זאת, למאקרו יש גם מספר חסרונות, כפי שנראה בהמשך.

### 3.5.2.1 פקודת מאקרו

יש הסוברים שפונקציית inline היא מאקרו, אך אין זה כך. הפרמטרים של מאקרו מטופלים באופן שונה מאלה של פונקציה. במיוחד יש לשים לב לעובדה שהמעבד העורך את המאקרו סורק את המקור והמהדר לא עושה זאת. כלומר, המהדר רואה כבר את תוצאת פעולת ההחלפה אשר מבצע המעבד של המאקרו.

נגדיר לדוגמה, את המאקרו הבא:

```
#define min(x, y)  ((x) < (y) ? (x) : (Y))
```

מאקרו זה מחזיר את הערך הנמוך ביותר שבין שני ערכים נתונים. כפי שכבר הסברנו, השימוש במאקרו דומה לשימוש בפונקציה. הנה דוגמה:

```
int x = -1;
unsigned y = 60;
int z = min(x, y);
```

התוצאה שהמשתמש מצפה לה היא ש-z מקבל את הערך 1-. אבל במספר מהדרים יומר הערך השלילי של המשתנה x לערך משתנה שלם חסֵר סימן, ולכן יקבל ערך הגבוה מ-60. הערך של המשתנה z הוא בלתי צפוי.

במקרה אחר ייתכן המצב הבא:

```
int x=1, y =2;
int z = min(x++, y);
```

במקרה זה, כוונת המשתמש היתה להגדיל את x **לאחר** מציאת המינימום בין x ל-y. השורה שנפרסת על ידי מעבד המאקרו היא כזו:

```
int z = ((x++) < (y) ? (x++) : y);
```

המשתנה x יגדל פעמיים, והתוצאה שיקבל z היא 2. למרות שהמאקרו נראה כפונקציה, הוא אינו פונקציה וגם ההתנהגות שלו שונה מזו של פונקציה. חיסרון אחר של מאקרו הוא שתוכניות ניפוי (debuger) אינן מכירות אותו. הסיבה לכך היא שהמהדר אינו יכול ליצור סמל עבורו בטבלת הסמלים, לכן אין כל אפשרות לדבג אותו.

**סמל** (symbol) הינו אלמנט אשר המהדר מחזיק עבור כל משתנה או פונקציה בטבלה, אשר קרויה **טבלת הסמלים** בזמן הידור.

פונקציות אשר אינן שייכות למחלקה יכולות אף הן להיות פונקציות **inline**. אפשר להגדיר פונקציה שמוצאת את הערך הנמוך גם באופן הבא:

```
inline int min(int x1, int x2)
{
      return (x1 < x2 ? x1 : x2);
}
```

פונקציה זו זהה מבחינת היעילות למאקרו הקודם שהגדרנו, אבל היתרון שלה בכך שהפעולות מתבצעות על הפרמטרים של הפונקציה פעם אחת בלבד. לכן השורה הבעייתית

```
int z = min(x++, y);
```

היתה מכניסה את הערך 1 למשתנה z ומקדמת את x ב-1 בלבד.

## 3.6 אופרטורים

**אופרטורים** הם פונקציות מיוחדות של המחלקה, אשר שמם כשם האופרטורים הבסיסיים של השפה. למשל, האופרטור גדול או קטן (>,<). אופרטורים יכולים להיות **אופרטורים בינארים** (binary operators), או **אופרטורים אונרים** (unary operators). אופרטורים בינארים הם אופרטורים בעלי שני אופרנדים ואופרטורים אונרים הם בעלי אופרנד אחד.

היכולת להגדיר אופרטורים ב-++C הופכת את מושג המחלקה לסוג חדש של משתנה המוגדר על ידי המשתמש. יכולת זו מאפשרת להרחיב את השפה. כלומר, להגדיר מחלקות אשר מתנהגות כמו משתנים וסוגים בסיסיים בשפה, לדוגמה שלמים. בסעיף זה נראה אופרטורים שונים שניתן להגדיר באמצעות מחלקות, וכך לדמותם למשתנים בסיסיים של השפה.

## 3.6.1 אופרטורים אונרים

**אופרטורים אונרים** (unary operators) מקבלים ארגומנט אחד. אופרטור כזה יכול להיות, למשל, אופרטור השלילה "!" של ++C (וגם של C). עבור מחלקה X ואופרטור op מגדירים אופרטור כלשהו (אונרי, או בינארי) בדרך הבאה:

```
class X {
//...
      ret-val operator op(arg-list);
};
```

לדוגמה, נגדיר את אופרטור השלילה עבור המחלקה String באופן הבא:

```
class String {
//...
public:
//...
      int operator!() const;
            { return (strlen(str)==0); }
};
```

האופרטור הוא **אונרי (**unary), משום שהוא פועל רק על האובייקט הנוכחי. במבט ראשון, נראה האופרטור קצת מוזר משום שאינו מקבל ארגומנטים. השימוש בו מסביר מדוע הוא אונרי:

```
String s("");

if (!s) {
      // ... do something
}
```

האופרטור פועל רק על האובייקט שלו, ונראה כאילו הוא מקבל ארגומנט אחד. במקרה זה האופרטור "!" יחזיר ערך אמת (אורך המחרוזת הוא אפס), ולכן יתבצע משפט if.

## 3.6.2 אופרטורים בינארים

**אופרטורים בינארים** (binary operators) הם כאלה הפועלים על שני ארגומנטים, לדוגמה האופרטור "<". אופרטור כזה מחזיר ערך השונה מאפס אם התנאי שהוא

אמור לבדוק נכון, אחרת יחזיר ערך אפס. למחלקה כמו String אפשר להגדיר מספר אופרטורים בינארים:

```
class String {
      char *str;
public:
//...
      int operator<(const String &s)
      { return (strcmp(str, s.str) < 0); }
      int operator==(const String &s)
      { return (strcmp(str, s.str) == 0); }
};
```

בקטע קוד זה מוגדרים שני אופרטורים. אחד עבור "קטן לקסיקוגרפית", ואחד עבור "שוויון". למרות ששני האופרטורים נראים כאילו הם מקבלים פרמטר אחד, הם פועלים למעשה על שני פרמטרים. הפרמטר הראשון סמוי - האובייקט הנוכחי, ואילו הפרמטר השני הוא האובייקט s שנמסר להם. השימוש בשני האופרטורים הוא כזה:

```
String s1("s1"), s2("s2");

if (s1 == s2)
      // something very wrong
else if (s1 < s2)
      // ok
```

כאשר מתבוננים בקטע הקוד שמשתמש בשני האופרטורים, קל להבין מדוע אלה **אופרטורים בינארים**, כי הם מקבלים שני אובייקטים. האובייקט השמאלי (s1) הוא האובייקט שעבורו הופעל האופרטור. במילים אחרות, זהו האובייקט שהמצביע **this** מצביע עליו. האובייקט הימני (s2) הוא האובייקט שמועבר כארגומנט לאופרטור.

אופרטור חשוב במיוחד הוא **אופרטור ההשמה** (assignment operator). לעיתים, אופרטור זה נקרא גם **אופרטור העתקה** (copy operator). עבור מחלקה X כלשהי ייראה אופרטור ההשמה כך:

```
X &operator=(const X &x);
```

כלומר, האופרטור מקבל ייחוס קבוע לאובייקט של X ומחזיר ייחוס לאובייקט של X. אופרטור זה חשוב במיוחד, משום שהוא מוענק אוטומטית על ידי המהדר, גם כאשר המתכנת אינו כותב אותו. ההעתקה המתבצעת על ידי האופרטור הזה היא **העתקה רדודה** (shallow copy), כמו במקרה של **בנאי ההעתקה** (copy constructor כלומר, רק השם מועבר ולא הקוד עצמו). במקרה של המחלקה String, לאחר העתקה כזו ישנם שני מצביעים אשר מצביעים לאותו אזור זיכרון:

```
String s1("s1");
String s2("");

s2 = s1;
```

כתוצאה ממצב זה ישוחרר אזור זיכרון זה פעמיים, פעם אחת על ידי פונקציית הפירוק של s1, ופעם אחת על ידי זו של s2. דבר זה יכול לגרום לתוצאות בלתי צפויות במחלקות שיש להן מצביעים, לכן יש להגדיר עבורן אופרטור העתקה. במקרה זה נגדיר את אופרטור ההעתקה באופן הבא:

```
class String {
//...
      String &operator =(const String &s);
};
```

ברגע שמוגדר אופרטור כזה, המהדר אינו מגדיר את אופרטור ברירת המחדל ואנו יכולים להגדיר את האופרטור שלנו שיישם את פעולת ההעתקה. הפעולה תתבצע על ידי הקצאת אזור זיכרון מתאים שלתוכו תועתק המחרוזת של האובייקט השני המשתתף במשפט ההשמה. אופרטור ההעתקה עבור המחלקה האחרונה ייראה כך:

```
String &String::operator = (const String &s)
{
      delete [] str;
      str = new char[strlen(s.str) + 1];
      strcpy(str, s.str);
}
```

**אופרטור העתקה** (copy operator) אינו יכול להיות פונקציה קבועה, מכיון שהוא משנה את האובייקט עליו הוא פועל.

**חוק:** למחלקה עם מצביעים מומלץ לכתוב אופרטור העתקה. דבר זה מומלץ גם למחלקות אשר העתקה רדודה אינה מתאימה להן.

## 3.6.3 סיכום האופרטורים לפי עדיפות

הטבלה הבאה מתארת את האופרטורים השונים של השפה ואת רמת העדיפות היחסית שלהם. האופרטור :: הוא בעדיפות 17, לכן הוא קודם לכל שאר האופרטורים. אופרטור הפסיק הוא בעל העדיפות הנמוכה ביותר.

האופרטורים: "sizeof", "::", ".", "?:" אינם ניתנים **להעמסה** (overloading, להגדרה מחודשת, או נוספת) על ידי מחלקה כלשהי. כל שאר האופרטורים ניתנים להעמסה. אופרטורים אשר לא תארנו עד עתה בפירוט יתוארו בהמשך הספר.

| אופרטור ועדיפות | | תיאור | דוגמה |
|---|---|---|---|
| :: | (17) | ציון שם מקומי, כמו שדה של מחלקה | `class_name::member` |
| :: | (17) | ציון שם גלובלי במרחב מקומי | `::name` |
| . | (16) | ציון שדה במחלקה או מבנה | `X obj; obj.mem` |

| | | | |
|---|---|---|---|
| -> | (16) | ציון שדה דרך מצביע | X *ptr; ptr->member |
| [] | (16) | ציון תא במערך | char arr[20];<br>arr[5] = 'a'; |
| () | (16) | קריאה לפונקציה | func(arg-list); |
| () | (16) | בניית אובייקט | X x(arg-list); |
| **sizeof** | (16) | החזרת גודל של אובייקט בבתים | X x; sizeof(X);<br>sizeof(x); |
| ++ | (15) | הגדלה באחד (post and pre) | int j; ++j; j++; |
| -- | (15) | הקטנה באחד (post and pre) | int j; --j; j++; |
| ~ | (15) | פעולת המשלים, הופך סיביות של מילה נתונה | int j,k;<br>j = ~k; |
| ! | (15) | פעולת השלילה, הופך ערכים לוגיים של אמת ושקר. | if (!j) { ... } |
| - | (15) | מינוס אונרי - שינוי סימן של משתנה | int j,k; j = -k; |
| + | (15) | פלוס אונרי | int j, k; j = +k; |
| & | (15) | לקיחת כתובת של משתנה | int j, *jp; jp = &j; |
| * | (15) | תוכן של מצביע | int j, *jp; j = *jp; |
| **new** | (15) | הקצאת זיכרון (גם של מערך בתוספת "[ ]") | char *s; X *xp;<br>s = new char [5];<br>xp = new X; |
| **delete** | (15) | שחרור זיכרון | char *s; X *xp;<br>delete [] s;<br>delete xp; |
| () | (15) | המרות של סוגים | int j; char c;<br>c = (char)j |
| .* | (14) | ציון שדה | struct X {<br>int x1, x2; };<br>int (X::*p); X x;<br>p = &x.x1;<br>x.*p = 1; |
| ->* | (14) | ציון שדה באמצעות מצביע | X *xp;<br>xp->*p = 1; |

| | | | |
|---|---|---|---|
| * | (13) | כפל | `int x1, x2, x3;`<br>`x1 = x2 * x3;` |
| . | (13) | חילוק | `int x1, x2, x3;`<br>`x1 = x2 / x3;` |
| + | (12) | חיבור | `int x, y, z;`<br>`z = x + y;` |
| - | (12) | חיסור | `int x,y,z;`<br>`x = y - z;` |
| >> | (11) | הזזה ימינה בסיביות. הזזה כזו גורמת לחלוקה בשתיים. | `int x, y;`<br>`y = x >> 2;` |
| << | (11) | הזזה שמאלה, שקולה להכפלה בשתיים | `int x,y;`<br>`x = y << 1;` |
| =<, < | (10) | קטן, קטן שווה, מתפתח לערכי אמת | `int x,y;`<br>`if (x < y) {...}` |
| =>, > | (10) | גדול, גדול שווה | `int x,y;`<br>`if (x >= y) { ... }` |
| == | (9) | שווה מקבל ערך אמת אם שני הצדדים זהים | `int x,y;`<br>`if (x == y) { ... }` |
| != | (9) | שונה (אינו שווה), מקבל ערך אמת, אם האופרנדים שונים | `int x,y;`<br>`if (x != y) {...}` |
| & | (8) | פעולת "וגם" על סיביות | `int x, y, z;`<br>`z = y & z;` |
| ^ | (7) | "או" אקסלוסיבי - סיביות שונות בלבד מקבלות ערך 1 | `int x, y, z;`<br>`z = x ^ y;` |
| \| | (6) | "או" של סיביות. מספיק שסיבית אחת נמצאת במקום המתאים, אז התוצאה היא אחד, בסיבית המתאימה. | `int x,y,z;`<br>`z = x \| y;` |
| && | (5) | "וגם" לוגי. כדי לקבל אמת, שני האופרנדים חייבים להיות אמת | `int x, y;`<br>`if (x && y) { ... }` |
| \|\| | (4) | "או" לוגי. | `int x,y;`<br>`if (x \|\| y) { ... }` |

| | | | |
|---|---|---|---|
| **? :** | (3) | תנאי | `int x,y,z;`<br>`z = (x>y ? x : y);` |
| = | (2) | פעולת השמה | `int x,y;`<br>`x = y;` |
| **oper=** | (2) | פעולה על האופרנד השמאלי. oper יכול להיות: "*", "/", "%", "+", "-", "<<", ">>", "&", "\|", "^". | `int x;`<br>`x *= 2;` |
| , | (1) | פעולה סדרתית. התוצאה של הביטוי האחרון בסדרה | `int x, y;`<br>`x = (y>1, 2);` |

בכל מקרה של ספק בעדיפות אופרטורים, אפשר להשתמש **בסוגריים** (כלומר '(...)'), כפי שאנו לציין את סדר הפעולות בביטויים מתמטיים.

## 3.7 שדות סטטיים המשותפים לאובייקטים

ב-C++ יש לכל אובייקט את השדות שלו ושדות אלה אינם משותפים למספר אובייקטים. אך לעיתים נגלה צורך בשדות אשר משותפים לכל האובייקטים. שדות המשותפים לכל האובייקטים נקראים, לעיתים, **שדות מחלקה** (class fields), או **שדות סטטיים** (static fields). לדוגמה, אם אנחנו רוצים לדעת את מספר האובייקטים ממחלקה מסוימת בכל זמן שהוא, היינו רוצים שדה המשותף לכל האובייקטים, שימָנֶה את מספר האובייקטים.

השדה הזה יכול להיות מוגדל בבנאי ומוקטן במפרק:

```
class X {
      static int xcount;
public:
      X() { xcount++; }
      ~X() { xcount--; }
};
```

שדה המשותף לכל האובייקטים יכול להיות מכל סוג בסיסי של השפה (או מחלקה אחרת), כאשר לפני שמו וסוגו מופיעה מילת המפתח **static**. שדה כזה אינו שייך לאובייקט מסוים, אלא לכל המחלקה. על כן, אין השדה נוצר כאשר יוצרים אובייקט מסוים. השדה קיים לפני כל האובייקטים כשהתוכנית מתחילה, והשדה נעלם (מפורק) רק כאשר התוכנית מסתיימת. לכן, יש לאתחל את השדה במקום כלשהו בתוכנית. אתחול כזה נעשה בצורה הבאה:

```
int X::xcount = 0;
```

מומלץ לאתחל שדות של מחלקה בקובץ המקור (source) של המחלקה. קבצים כאלה כוללים את פונקציות המחלקה ומסתיימים בדרך כלל ב-C או CPP.

## 3.7.1 פונקציות סטטיות

פונקציות הפועלות על אובייקטים מקבלות ארגומנט, או פרמטר נוסף בצורה לא מפורשת. ארגומנט זה הוא המצביע לאובייקט הנוכחי. ארגומנט זה מוענק באופן אוטומטי לכל פונקציה החברה במחלקה על ידי המהדר. מילת המפתח **this** היא הארגומנט החבוי הזה.

עבור מחלקה כלשהי X ופונקציה כלשהי השייכת למחלקה, אפשר להתייחס למצביע **this** בתוך הפונקציה כמצביע לאובייקט הנוכחי אשר עליו פועלת הפונקציה. למשל:

```
class X {
      int xval;
public:
      X(int x) { xval = x; }
      int get() const { return xval; }
};
```

לפונקציה get יש פרמטר חבוי אשר מצביע לאובייקט הנוכחי שדרכו מוחזר הערך של השדה xval באובייקט הנוכחי. הפונקציה get יכולה להיות כתובה גם באופן הבא, והיא שקולה לגירסה הקודמת:

```
int get() const { return this->xval; }
```

לפונקציות סטטיות אין מצביע לאובייקט הנוכחי, מכיון שאין להן אחד כזה. לפונקציה סטטית יש גישה ל**חלק הפרטי** של **המחלקה**, אבל כדי לגשת לחלק הפרטי של **אובייקט מסוים**, יש להעביר לפונקציה מצביע לאובייקט.

לעומת זאת, אם היא פונה לשדות סטטיים של האובייקט היא אינה צריכה מצביע לאובייקט, היות והשדה הסטטי אינו שייך לאובייקט מסוים. למשל:

```
class X {
      int val;
      static int count;
public:
      X(int v) { val = v; count++;}
      ~X() { count--; }
      static int getObjectsCount() { return count; }
      static int getVal(X *xptr) { return xptr->val; }
      int get() { return val; }
};
```

במחלקה זו יש שני שדות. אחד (val) רגיל, והשני (count) סטטי. למחלקה יש שתי פונקציות סטטיות: הראשונה (getObjectsCount) מחזירה את מספר האובייקטים הנוכחיים במחלקה. פונקציה זו אינה צריכה לקבל מצביע לאובייקט. הפונקציה הסטטית השנייה, אשר מחזירה מצביע לשדה של האובייקט, צריכה מצביע לאובייקט כדי לגשת לשדה המבוקש.

הפונקציה השלישית של המחלקה (get) אינה פונקציה סטטית, ולכן אינה צריכה מצביע לאובייקט משום שזה האחרון מסופק אוטומטית על ידי המהדר. הפונקציה יכולה לגשת באופן ישיר לכל שדה של האובייקט, וגם לשדות המוגדרים כשדות סטטיים.

## 3.7.2 דוגמה לשימוש בשדות ופונקציות סטטיות

כדי להמחיש את השימוש במנגנון של שדות ופונקציות סטטיות נעיין בדוגמה. נתונה לנו מערכת מדפסות. מותר להגדיר שלוש מדפסות לכל היותר בעת ובעונה אחת. כדי לענות על דרישה זו נגדיר מחלקה Printer שמייצגת מדפסת.

```
class Printer {
    static int count;
    int fileDes;
    Printer() { fileDes = openprt();  count++; }
    Printer(const Printer &);
public:
    ~Printer() { closeprt(fileDef); count--; }
    static Printer *createPrinter()
    { return (count < 3 ? new Printer() : 0); }
    int print(const char *str);
};
```

למחלקה Printer יש שדה (count) הסופר את מספר האובייקטים הקיימים בכל זמן ושייכים למחלקה. נניח, שהקשר למדפסת הוא דרך (file-descriptor) של UNIX, שהוא מספר שלם. כמו כן, נניח שיש פונקציות של המערכת המאפשרת פתיחה של ערוץ למדפסת וסגירה של ערוץ כזה (openprt, closeprt). לכל מדפסת יש מספר המאפיין אותה במערכת ההפעלה. נמקם את הבנאים של המחלקה בחלקה הפרטי, כדי למנוע מהמשתמש במחלקה זו להגדיר מספר אובייקטים בלתי מוגבל של המחלקה. כתוצאה מכך, אין המשתמש במחלקה יכול להגדיר אובייקטים באופן הבא:

```
Printer prt;
```

כדי להגדיר אובייקטים ממחלקה זו, עליו להשתמש בפונקציה createPrinter. פונקציה זו בודקת אם מספר האובייקטים קטן משלושה. אם כן, מייצרת אובייקט חדש ומחזירה מצביע אליו, אם יש יותר משלושה אובייקטים מחזירה הפונקציה מצביע שערכו אפס. מצביעים מאופסים ב-C++ שקולים ל-NULL.

המשתמש משתמש בפונקציה createPrinter באופן הבא:

```
Printer *prt = Printer::createPrinter();
if (prt)
      prt->print("A string");
```

הקריאה לפונקציה סטטית נעשית על ידי צירוף של המרכיבים האלה: המחלקה, לאחריה "::" ולאחר מכן שם הפונקציה, כמו שראינו בדוגמה למעלה.

## 3.8 מנגנון החבר (friend)

מנגנון המחלקה מאפשר הסתרת מידע ממשתמשי המחלקה. הסתרת המידע נועדה להגן על המשתמש במחלקה מפני שינויים במבנה הנתונים הפנימי שלה. שינויים אלה לא צריכים להשפיע על משתמשי המחלקה, אלא על פונקציות המחלקה בלבד.

ייתכנו מקרים בהם נרצה לאפשר לפונקציות, או למחלקות אחרות, גישה לחלק הפרטי של המחלקה. באופן עקרוני דבר זה **אינו מומלץ** ויש להשתמש במנגנון זה רק כאשר אין אפשרות אחרת להשגת פונקציונליות רצויה.

כדי לאפשר גישה למחלקה, או לפונקציה אחרת, יש לציין את הדבר במחלקה שמאפשרת זאת. לדוגמה, נניח שמחלקה X מאפשרת לפונקציה f ולמחלקה Y את הגישה לחלק הפרטי שלה. המחלקה X מציינת את חבריה, המחלקה Y והפונקציה f, בעזרת מילת המפתח **friend**. לדוגמה:

```
class X {
      friend class Y;
      friend void f(X*);
public:
//..
};
```

במקרה הראשון יש גישה לכל החברים (פונקציות) של מחלקה Y לשדות הפרטיים של X. כמו כן יש גישה לחלקים הפרטיים של X על ידי הפונקציה f.

### 3.8.1 מנגנון החבר בשימוש

כעת נחזור למחלקה String ונוסיף לה מנגנונים ואופרטורים נוספים אשר ישתמשו במנגנון החבר. בחלקה הפרטי של המחלקה נגדיר שתי פונקציות שמעדכנות את המחרוזת שמחלקה זו מכילה. פונקציות אלו ישמשו את פונקציות המחלקה, או חברים של המחלקה.

הפונקציה הראשונה שמקבלת פרמטר אחד, מצביע לתווים, מעדכנת את התכולה לערך מצביע זה. הפונקציה השנייה מקבלת שתי מחרוזות ומשרשרת ביניהן. שאר הפונקציות של המחלקה הן פונקציות **inline** המשתמשות בפונקציות עדכון המצביע str של המחלקה String.

שתי הפונקציות המוזכרות הן בעלות אותו שם (set). דבר זה חוקי ב-C++ ונקרא **overloading** או **העמסה** בעברית. ההבחנה בין פונקציות אלו נעשית במקום הקריאה לפונקציה, על פי הארגומנטים שמעבירים לפונקציה.

המהדר אינו משתמש בערך המוחזר מהפונקציה כדי להבדיל בין פונקציות מועמסות. אם יש שתי פונקציות מועמסות השונות ביניהן אך ורק בערך המוחזר, המהדר יציין זאת כשגיאת הידור. חוק זה נובע מהעובדה שקריאה לפונקציה אינה חייבת להשתמש בערך המוחזר על ידה. לכן, אם יש שתי פונקציות השונות רק בערך המוחזר, אין אפשרות להבדיל ביניהן במקרה כזה.

### 3.8.1.1 בנאים ל-String

לאחר הגדרת הפונקציות הפרטיות, מוגדרים הבנאים של המחלקה בחלק הציבורי שלה. הבנאי הראשון מקבל מצביע למחרוזת ויוצר אובייקט מסוג String. בנאי זה משתמש בגירסה הראשונה של הפונקציה set.

הבנאי מאפס את המצביע str לפני גוף הבנאי. משפטים כאלה, המתרחשים לפני גוף הבנאי ומופיעים מייד לאחר ":" ולפני הסוגריים המסולסלים, מסמנים את תחילת גופו של הבנאי (כלומר, הקוד של הפונקציה). אם יש מספר שדות המאותחלים לפני גוף הבנאי, מופיעים אלה אחד לאחר השני כאשר פסיק מפריד ביניהם.

בנאי זה נקרא לעיתים גם **אופרטור הבנאי** (construction operator). בנאי המקבל פרמטר אחד יכול לשמש גם להמרה מסוג הפרמטר לאובייקט של המחלקה. לדוגמה, אם נתונה פונקציה שמקבלת אובייקט מסוג String, וקוראים לה עם מצביע למחרוזת, יבצע המהדר המרה אוטומטית של המצביע לאובייקט של המחלקה בעזרת בנאי ההמרה. לדוגמה:

```
void f(String s)
{
      // ...
}

f("C string");
```

לפני הקריאה לפונקציה ייצור המהדר אובייקט זמני מסוג String מהמחרוזת, ויקרא לפונקציה עם אובייקט זמני זה. המהדר עצמו ידאג לפרק את האובייקט לאחר הקריאה לפונקציה.

הבנאי השני יוצר אובייקט מסוג String המכיל מצביע למחרוזת, שהיא שרשור של שתי המחרוזות המועברות אליו. בנאי זה משתמש בגירסה השנייה של הפונקציה set.

הבנאי השלישי הוא בנאי ההעתקה. בנאי זה מוענק למחלקות שאינן מספקות אותו כבנאי ברירת המחדל. כשהמהדר מעניק בנאי זה, הוא משתמש בשיטת העתקה רדודה. היות ויש לנו מצביע במחלקה, עלינו לספק בנאי העתקה.

המפרק של המחלקה משחרר את הזיכרון אשר הוקצה באחת מפונקציות set. השחרור נעשה זאת בעזרת קריאה לאופרטור **delete** של מערך.

```
class String {
  char *str;
  String *set(const char *s);
  String *set(const char *s1, const char *s2);
public:
  // constructors
  String(const char *s) : str(0) { set(s); }
  String(const char *s1, const char *s2) : str(0)
  { set(s1, s2); }
```

```
  String(const String &s) : str(0) { set(s.str); }
  ~String() { delete [] str; }

  // assignment operators
  String &operator=(const String &s){ return *set(s.str);
}
  String &operator=(const char *s) { return *set(s); }

  // equality operators
  int operator==(const String &s)
  { return (strcmp(str, s.str)==0); }
  friend int operator==(const char *str, const String &s)
  { return (strcmp(str, s.str) == 0); }

  // lexical operators
  int operator<(const String &s)
  { return (strcmp(str, s.str) < 0); }
  friend int operator<(const char *str, const String &s)
  { return (strcmp(str, s.str) < 0); }

  // concatenation of two strings
  friend String operator+(const char *s1, const String &s2)
  { return String(s1, s2.str); }
  String operator+(const String &s)
  { return String(str, s.str); }
  String &operator+=(const String &s)
  { return *set(str, s.str); }

  // accessors
  const char *get_cstr() const { return str; }
};
```

### 3.8.1.2 אופרטורים

למחלקה String יש שני **אופרטורי השמה** (assignment operators), אשר קרויים לעיתים גם בשם **אופרטורי העתקה** (copy operators). האופרטור הראשון מקבל ייחוס לאובייקט מסוג String. הוא יוצר אובייקט המכיל מצביע למחרוזת הזהה למחרוזת עליה מצביע האובייקט הנתון. האופרטור מחזיר **ייחוס** (reference) לאובייקט של מחלקה מסוג String. דבר זה נעשה כדי לאפשר מספר השמות בזו אחר זו:

```
String s1("s1"), s2("s2");
String s3("");

s3 = s1 = s2;
```

משפט ההשמה המורכב מאפשר השמה של s2 ל-s1 וגם ל-s3. גם במקרה זה הייחוס שימושי. האובייקט המוחזר מאופרטור ההשמה הוא ייחוס, ולכן נמנעות העתקות מיותרות של אובייקטים.

האופרטור השני מאפשר השמה של מחרוזת רגילה לאובייקט מסוג String. אפשר להשתמש באובייקטים אלה בדרך זו:

```
s3 = "A new string";
```

**אופרטורי ההשוואה** (comparison operators) מאפשרים להשוות בין שני אובייקטים מסוג String, או בין מחרוזת לאובייקט מסוג String. אופרטור ההשוואה השני מוגדר כחבר של המחלקה. אופרטור זה אינו פונקציה של המחלקה. הסיבה לכך היא, שאופרטור שהוא פונקציה של מחלקה חייב לקבל אובייקט בצידו הימני. לכן, אם אנו רוצים לכתוב למשל:

```
if ("A new string" == s3)
      //...
```

אנחנו חייבים להגדיר את אופרטור ההשוואה כחבר (friend) של המחלקה. מומלץ שלא להגדיר פונקציות כחברות של מחלקה, משום שזה פוגע בהסתרת המידע. יש להימנע מהגדרת פונקציות כאלו, פרט למקרים מיוחדים, כמו המקרה לעיל.

האופרטור ההפוך, המקבל מצביע למחרוזת, אינו מחויב המציאות, מכיון שיש בנאי המאפשר המרה של מצביעים למחרוזות, לאובייקטים של String. המהדר יודע ליצור אובייקט זמני במקרים אלה, לאחר מכן המהדר משתמש באובייקט הזמני כדי לקרוא לאופרטור ההשוואה הראשון. באופן דומה מוגדרים אופרטורי הסדר הלקסיקוגרפי של אובייקטים מסוג String.

לבסוף, מוגדרים האופרטורים לשרשור בין שתי מחרוזות. האופרטור הראשון הוא חבר של המחלקה ומאפשר לשרשר בין מחרוזת של C לבין אובייקט מסוג String. הפרמטר הראשון של אופרטור זה הוא מחרוזת של C.

האופרטור השני מאפשר לשרשר בין שני אובייקטים מסוג String. בעזרתו אפשר לכתוב:

```
String s1("..."), s2("..."), s3("...");
s3 = s1 + s2;
```

**אופרטור חיבור-השמה** (plus-assignment operator) האחרון מאפשר שרשור של אובייקט אחד לשני, ולהשתמש בתוצאה. דוגמה:

```
s1 += s2;
```

שתי הפונקציות, המבצעות את רוב העבודה, הן set. הן מאפשרות לקבוע את הערך של אובייקט מסוג String למחרוזת אחת, או לשתיים.

הגירסה הראשונה של הפונקציה set מקבלת מצביע לתו. הפונקציה מקצה מערך של תווים הגדול באחד מאורכה של המחרוזת הנתונה s. הפונקציה מעתיקה את המחרוזת שהיא מקבלת (s) לאזור הזיכרון החדש ולאחר מכן, משחררת את אזור הזיכרון הישן. הפונקציה מחזירה מצביע לאובייקט הנוכחי. הדבר מאפשר להפעיל את הפונקציה

ברציפות מספר פעמים. עובדה זו מנוצלת על ידי האופרטור "=" המחזיר ייחוס (reference) לאובייקט הנוכחי כדי לאפשר השמות כפולות של אובייקטים מסוג String.

```
String *String::set(const char *s)
{
	char *newStr = new char[strlen(s) + 1];
	strcpy(newStr, s);
	delete [] str;
	str = newStr;
	return this;
}
```

הגירסה השנייה של הפונקציה set מקבלת שני מצביעים למערכים של תווים. הפונקציה מקצה זיכרון המספיק להכיל את שתי המחרוזות, בתוספת של תו אפס המסמן את סוף המחרוזת. לאחר מכן, משרשרת הפונקציה את שתי המחרוזות ומחזירה מצביע לאובייקט הנוכחי. עובדה זו מנוצלת על ידי האופרטור "+=". אופרטור זה קורא לפונקציה זו ומחזיר ייחוס לאובייקט הנוכחי. האופרטור מחזיר את תוכן המצביע המוחזר על ידי הפונקציה set. עובדה זו מאפשרת להפעיל את האופרטור הזה ברציפות באותו משפט.

```
String *String::set(const char *s1, const char *s2)
{
	int l1 = strlen(s1), l2 = strlen(s2);
	char *newStr = new char[l1 + l2 + 1];
	strcpy(newStr, s1);
	strcpy(newStr + l1, s2);
	delete [] str;
	str = newStr;
	return this;
}
```

לפיכך, לאחר פעולת הפונקציה האחרונה מכיל האובייקט הנוכחי את השרשור של שתי המחרוזות הניתנות לפונקציה.

קטע הקוד הבא מדגים את השימוש במחלקה זו:

```
int main()
{
	String s1("s1"), s2("s245"), s3("s3");
	String s4 = s1;

	s4 = s1 + s2;
	if ("s1" == s1)
		cout << "OK s1 == s1" <<  endl;
	if ("s0" < s1)
		cout << "OK s0 < s1" << endl;
```

```
      cout << "s4 = " << s4.get_cstr() << endl;
      return 0;
}
```

בקטע קוד זה מוגדרים מספר אובייקטים מסוג String, ולאחר מכן מופעלים האופרטורים השונים. קטע הקוד מראה שהשימוש באובייקטים מסוג זה דומה לשימוש באובייקטים בסיסיים של C++, כגון שלמים, תווים ומספרים ממשיים. המחלקה ב-C++ היא סוג חדש המרחיב את שפת המתכנת.

המחלקה String יכולה גם להיכתב בעזרת פונקציות ההקצאה והשחרור של C, כדי להשיג יעילות גבוהה יותר. אפשר להשתמש בפונקציה realloc ובפונקציה free כדי לאפשר שימוש חוזר באותו אזור זיכרון, במקום להקצות מחדש את הזיכרון בכל פעם שמשרשרים מחרוזות או מבצעים השמה. בדרך כלל, מומלץ להשתמש באופרטורים המתאימים, new ו-delete, של C++.

**הערה**: הפונקציה **realloc** מאפשרת לשנות את גודל הזיכרון שהוקצה על ידי malloc. הפונקציה **free** משחררת אזור זיכרון שהוקצה על ידי malloc או על ידי realloc.

# 3.9 אובייקטים כשדות של אובייקטים אחרים

לעיתים יש צורך ליצור אובייקטים המורכבים ממספר אובייקטים אחרים. אם נרצה לנהל מערכת שאוגרת ומבצעת פעילויות שונות של אנשים, יהיה לנו אובייקט שיתאר אדם. לכל אדם במערכת כזו יש שם פרטי ושם משפחה, ואולי גם פרטים נוספים. אם לפנינו מערכת לניהול שכר, יכולה להיות לאובייקט מסוג זה משכורת חודשית ושעות עבודה חודשיים.

## 3.9.1 הגדרת תת-אובייקטים

נתייחס למערכת ניהול שכר כמתואר לעיל ונגדיר אובייקט המייצג אדם במערכת זו. השם הפרטי ושם המשפחה הם מחרוזות, ואנו רוצים להגדירם כאובייקטים של המחלקה String. דבר זה ניתן לעשות ב-C++ בצורה הבאה:

```
class Person {
  String name;          // the name of the person
  String surName;       // family name
  int salary;           // the salary
  int hours;            // hours for last month
public:
  Person(const char *nm, const char *sn, int sal, int hrs);
  void rename(const char *nm)
  { name = nm; }
  int operator==(const Person &);
};
```

השדות name ו-surName הן השם ושם המשפחה של אדם הנמצא במערכת. שני שדות אלה הם אובייקטים של המחלקה String. שדות אלה הם **תת-אובייקטים** באובייקט מסוג Person. פרט לכך, מוגדרים שדות נוספים שמייצגים את המשכורת החודשית ואת מספר השעות שאדם זה עבד בחודש האחרון.

הבנאי של מחלקה זו מקבל ארבעה פרמטרים אשר מייצגים את הערכים ההתחלתיים שיש לאובייקט ממחלקה זו. תבנית פונקציית הבנאי של המחלקה נראית כך :

```
Person::Person(const char *nm,const char *sn,int sal,int hrs)
      : name(nm), surName(sn)
{
      salary = sal;
      hourse = hrs;
}
```

לפני גוף הבנאי מופיע רשימה של אתחול השדות שהם אובייקטים. במקרה זה, יש שני שדות כאלה : name ו-surName. האתחול נעשה בצורה דומה להגדרה של משתנים רגילים מהמחלקה String. הבנאי הראשון של המחלקה String מופעל ומקבל מחרוזת שהיא השם, או שם המשפחה. רשימת האתחול של השדות, שהם אובייקטים של מחלקות אחרות, מתחילה מהאופרטור ":" ומסתיימת בסוגר המסולסל השמאלי ("{") המסמן את תחילת גוף הבנאי. אתחול שדות אלה נכתב לפני גוף הבנאי.

סדר האתחול, או ביצוע הבנאים של תת-האובייקטים הוא לפי סדר ההגדרה שלהם במחלקה. במקרה זה יאותחל תמיד השם (name) לפני שם המשפחה (surName), גם אם סדר שדות אלה ברשימת האתחול של הבנאי יהיה הפוך.

בגוף הבנאי מאותחלים השדות האחרים, שהם מבנים בסיסיים של השפה, כגון המשכורת החודשית. סדר אתחול זה הוא לפי סדר המשפטים בגוף הבנאי.

למרות שלא הוגדרה פונקציית פירוק למחלקה Person, תוגדר כזו למחלקה באופן אוטומטי על ידי המהדר. סדר הפירוק של תת-אובייקטים יהיה הפוך לסדר האתחול. ראשית, ייקרא המפרק של האובייקט המורכב, במקרה זה Person, ולאחר מכן ייקראו המפרקים של האובייקטים surName ו-name לפי סדר זה.

הפונקציה rename משתמשת באופרטור ההשמה של המחלקה ממחרוזת כדי לשנות את השם הנוכחי של אובייקט מסוג Person.

### 3.9.2 העתקת אובייקטים מורכבים

כזכור, לכל אובייקט יש שתי פונקציות אשר מוגדרות באופן אוטומטי על ידי המהדר. לעומת זאת, יכול המתכנת להגדיר פונקציות אלו עצמו ולדרוס את גרסת המהדר. פונקציות אלו הן **בנאי ואופרטור העתקה**. הגירסה של פונקציות ההעתקה שמייצר המהדר, מבצעת את בנאי או אופרטורי ההעתקה של תת- האובייקטים. לפיכך, במקרה זה לא נזקקנו לאופרטור העתקה.

אם נרצה לספק בנאי העתקה (או אופרטור העתקה) למחלקה זו, נכתוב זאת כך :

```
Person::Person(const Person &per)
      : name(per.name), surName(per.surName)
{
      salary = per.salary;
      hours  = per.hours;
}
```

גירסה זו של בנאי העתקה שקולה לגירסה המסופקת על ידי המהדר כ**ברירת מחדל** (default option). במצב שאין סמנטיקה של העתקה עמוקה, אין צורך להגדיר את הפונקציות האלו (בנאי ואופרטור העתקה) שמיוצרות באופן אוטומטי על ידי המהדר.

## 3.9.3 דוגמה לשימוש במצביעים לתת-אובייקטים

בפרק זה ראינו איך להגדיר מחלקה המגבילה את מספר המופעים של אובייקטים ממחלקה זו. לדוגמה, המחלקה Printer הגבילה את מספר המופעים של אובייקטים ממחלקה זו לשלושה אובייקטים, לכל היותר. בסעיף זה נשיג את אותה התוצאה תוך שימוש במחלקה החברה למחלקה אחרת.

נגדיר שתי מחלקות. ראשונה מבצעת את הפעילות האמיתית על המדפסת. מחלקה זו נקראת Printer. המחלקה השנייה משמשת **כמעטפה** (envelop) למחלקה הראשונה ותיקרא PrinterEnv. תאפשר ליצור אובייקטים של Printer.

```
class Printer {
      friend class PrinterEnv;
      int fileDes;      // connection to printer
      Printer();
public:
      int print(const char *str);
};

class PrinterEnv {
      static int prtCount;
      Printer *prt;     // the actaul printer
public:
      PrinterEnv() {
            count++;
            if (count < 3) prt = new Printer;
            else prt =0;
      }
      ~PrinterEnv() { count--; delete prt; }
      Printer *get() { return prt; }
};
```

המחלקה PrinterEnv היא חברה של המחלקה Printer. למחלקה PrinterEnv יש שדה סטטי המציין כמה אובייקטים ממחלקה זו קיימים. הבנאי של מחלקה זו יוצר אובייקט מסוג Printer רק אם מספר האובייקטים קטן משלושה. מאחר והבנאי של Printer נמצא בחלק הפרטי, אפשר ליצור אובייקטים מסוג זה רק דרך PrinterEnv שמגבילה את מספר האובייקטים מסוג Printer.

## 3.10 הגדרה מקוננת של מחלקות

C++ תומכת ב**הגדרה מקוננת** (nested declaration) של מחלקות. כלומר, ניתן להגדיר מחלקה בתוך הגדרה של מחלקה אחרת (nesting, **קינון** מלשון 'קן'). מחלקה שמוגדרת בתוך מחלקה אחרת אינה מעמיסה את מרחב השמות הגלובלי ואינה גורמת להתנגשויות עם מחלקות אחרות בעלות אותו שם. שם אחר למחלקה מקוננת הוא מחלקה פנימית. תחום **המחלקה הפנימית** הוא תחומה של **המחלקה החיצונית**. למשל:

```
class Line {
    class point {
      int x,y;
    public:
      point(int xo, int yo) { x=xo; y=yo }
      int getX() const { return x; }
      int getY() const { return y; }
      int equal(const point &);
    };
    point p1, p2;
public:
    ...
};
```

המחלקה point מוגדרת בתוך המחלקה Line, בחלק הפרטי שלה. במקרה זה, המחלקה הפנימית תהיה מוכרת לשאר המחלקות, או הפונקציות, רק אם היא בחלק הציבורי של המחלקה החיצונית. מחלקות המוגדרות בחלק הפנימי מוכרות רק לפונקציות של המחלקה החיצונית, או מחלקות, או פונקציות אחרות המוגדרות כחברות של המחלקה החיצונית.

לפיכך, המחלקה point מוכרת אך ורק לפונקציות המחלקה Line. לפונקציות המחלקה Line יש זכות גישה לחלק הציבורי בלבד של המחלקה point.

ראינו שרוב הפונקציות של המחלקה point מוגדרות בגוף המחלקה. לעיתים, יש צורך להגדיר פונקציות כאלו מחוץ למחלקה. במקרה כזה יש לציין שהפונקציה היא של מחלקה המוגדרת במחלקה אחרת באופן הבא:

```
int Line::point::equal(const point &p)
{
// ...
}
```

שם המחלקה המכילה מופיע לפני שם המחלקה המקוננת, בתוספת אופרטור התחום "::". בתוך הפונקציה, וגם ברשימת הפרמטרים, ההתייחסות היא לאובייקטים מסוג point של המחלקה המקוננת. בדוגמה זו מודגש שהמחלקה המקוננת הינה מחלקה פנימית, וכך היא גם נקראת.

כעת נוכל להגדיר, בחלק אחר של התוכנית, מחלקה point באופן הבא:

```
class point {
      int x, y;
public:
      point(int xo=0, int yo=0)
      { x=xo; y=yo; }
      int get_x() const {return x;}
      int getY() const { return y; }
};
point p1;
```

המשפט האחרון מגדיר אובייקט מסוג point, אבל לא של המחלקה המקוננת, ולכן אפשר להשתמש בו באופן הבא:

```
int x=p1.get_x();
int y=p1.getY();
```

אם, לעומת זאת, היתה המחלקה המקוננת מוגדרת בחלק הציבורי של המחלקה Line, ניתן היה להגדיר אובייקטים של מחלקה זו כך:

```
Line::point p1(1,2), p2(3,4);
```

הפונקציות המופעלות על אובייקטים אלה תהיינה פונקציות של המחלקה המקוננת ולא של המחלקה הגלובלית.

מומלץ להשתמש בתכונה זו כאשר מגדירים ספריות, מבלי "לזהם" את מרחב השמות הגלובלי. אם הספריות הן גנריות, ולשימוש נרחב, יש סיכויים להתנגשויות של שמות עם ספריות אחרות.

## 3.11 רשומות כמחלקות

ב-++C יש אפשרות להגדיר רשומות struct כמו ב-C. ההגדרה הבאה היא חוקית:

```
struct point {
      int x,y;
};

point p;
```

שלא כמו ב-C, אין צורך להוסיף את מילת המפתח struct כאשר מגדירים אובייקטים (מופעים) של הרשומה. ב-++C יש שתי תוספות נוספות: אפשרות להגדיר אזורי גישה, ואפשרות להגדיר פונקציות החברות ברשומה.

## 3.11.1 אזורי גישה

ברירת המחדל של אזורי הגישה ברשומות היא **גישה ציבורית**. כלומר, לכל הפונקציות יש גישה לשדות הרשומה. אפשר לשנות את הגישה של שדות בעזרת שימוש במילות המפתח **private** או **public**. שימוש במילות מפתח אלו מגדיר אזור גישה פרטי או ציבורי, בהתאמה, עד לשימוש הבא במילת מפתח אחרת. לדוגמה:

```
struct point {
      int x;
public:
      int y;
private:
      int z;
};

void f(point &p)
{
      p.x = 1;    // ok - public by default
      p.y = 2;    // ok - public
      p.z = 3;    // error - private!!!
}
```

הפונקציה f אינה יכולה לגשת ל**שדה הפרטי** z משום שהוא בחלק הפרטי של הרשומה. לעומת זאת, היא יכולה לגשת לשדות האחרים של המחלקה, משום שהם בחלק הציבורי. יש אפשרות לשנות את **אזורי הגישה** מספר לא מוגבל של פעמים.

## 3.11.2 פונקציות של רשומות

אפשר להגדיר שדות פרטים לרשומה, ולכן אך טבעי הדבר שתהיה אפשרות להגדיר פונקציות הפועלות על רשומות. ב-C++ רשומה היא מחלקה לכל דבר. אפשר להוסיף פונקציות המטפלות בשדות הפרטים של point בצורה הבאה:

```
struct point {
      int x;
public:
      int y;
private:
      int z;
public:
      void setZ(int zo) { z = zo; }
      int getZ() const { return z; }
};
```

שתי הפונקציות בחלק הציבורי של הרשומה מאפשרות טיפול בשדה הפרטי z של הרשומה. לפונקציות כאלה יש מצביע מוסתר (**this**) לאובייקט הנוכחי, בדומה לפונקציות של מחלקה, והשימוש בהן הוא כזה:

```
point p;
p.setZ(5);
int z = p.getZ();
```

לאחר סדרת משפטים זו יקבל המשתנה z את הערך 5. לכן, רשומות הם אובייקטים רגילים ב-C++. ההבדל בין רשומה למחלקה הוא שאזורי הגישה הם ציבוריים, אלא אם כן צוין אחרת.

# 3.12 מחלקות חסכוניות

C++ תומכת בשני מושגים המאפשרים לדחוס זיכרון: **איחוד** (union) **ושדה סיביות** (bitfield). **האיחוד** שימושי במקרה שיש מספר משתנים, ורק אחד מהם פעיל בכל רגע נתון. במקרה כזה, ניתן לארוז את המשתנים באזור זיכרון אחד. **שדה סיביות** שימושי כאשר יש מספר משתנים המקבלים ערכי אמת (true) או שקר (false) בלבד.

הדוגמאות המופיעות בסעיף זה נמצאות בקובץ **spclcs.cpp** בדיסקט המצורף לספר.

## 3.12.1 איחוד (union)

**איחוד** (union) הוא מבנה המאפשר לקבץ מספר משתנים באזור זיכרון משותף. גודל המבנה המאוחד הוא כגודלו של המשתנה הגדול ביותר. הדבר מצמצם את אזורי הזיכרון המאחסנים אוספי שדות. הדרך להגדרת איחוד היא זו:

```
union <union-name> {
members;
};
```

איחוד הוא מחלקה שכל חבריה הם ציבוריים, לפי ברירת המחדל. גישת שדות האיחוד ניתנת לשינוי על ידי מילות המפתח: **public**, **protected** ו-**private**.

אפשר להגדיר את האיחוד הבא:

```
union X { char c; int x; };
```

גודלו של אובייקט מסוג X יהיה זהה לגודל של שלם (int). גודל זה תלוי, כמובן, במחשב עליו מופעלת התוכנית. אובייקט מסוג X מכיל שני שדות החופפים באזור הזיכרון: x ו-c. למהדר אין שום אפשרות לדעת איזה ערך נמצא כרגע באיחוד, לכן האחריות מוטלת על המתכנת. לדוגמה:

```
X x;
char c;
int j;
```

```
x.x = j;
c = x.c;
```

ההשמה בשורה האחרונה תגרום לאיבוד מידע במקרה הטוב, ובמקרים אחרים להפסקת התוכנית. מכך ניתן להסיק שהשימוש באיחודים אינו פשוט כלל ועיקר.

הטכניקה אותה מפעילים, בדרך כלל, היא: להכניס איחוד כזה לתוך מחלקה השומרת את הסוג הנוכחי של האובייקט שנמצא באיחוד. המחלקה הבאה (Union) מדגימה טכניקה זו.

```
class Union {
public:
        enum value_type { int_type, char_type,
                          float_type, double_type};
private:
        union X {
                char cval;
                int ival;
                float fval;
                double dval;
        };
        X x;
        value_type type;
public:
        Union(char c)
        { x.cval = c; type = char_type; }
        Union(int i)
        { x.ival = i; type = int_type; }
        Union(float f)
        { x.fval = f; type = float_type; }
        Union(double d)
        { x.dval = d; type = double_type; }
        int get(char &c)
        { c = x.cval; return (type == char_type); }
        int get(int &v)
        { v = x.ival; return (type == int_type); }
        operator float()
        { return x.fval; }
};
```

**האיחוד** המוגדר במחלקה מכיל מספר שדות שמתחלקים באותו אזור זיכרון בו-זמנית. המחלקה שמכילה את האיחוד, מחזיקה שדה נוסף שמסמן את סוג המשתנה הפעיל בכל רגע באיחוד. שדה זה מוגדר כ-enum. בנוסף, המחלקה העוטפת את האיחוד מגדירה בנאים המקבלים ערכים בהתאם לערך של האיחוד. במקרה זה, למשל, מוגדרים בנאים עבור ערכים שלמים וערכים מסוג float.

כדי לקבל את הערך של האיחוד מוגדרות מספר פונקציות גישה. כל פונקציה מחזירה ערך אמת (ערך השונה מאפס, בדרך כלל אחד) אם האיחוד מתאים לערך המבוקש. לדוגמה: הפונקציה get הראשונה מחזירה ערך שלם בייחוס שהיא מקבלת, ואינדיקציה המסמנת לפונקציה שקראה לה אם הערך הנוכחי מתאים לערך המבוקש (במקרה זה - שלם).

## 3.12.2 שדות סיביות (bits)

לשדות מסוימים יש ערכי אמת בלבד, אפס או אחד. במקרים כאלה אפשר להציג את הערכים במשתנה מסוג **char**. אם יש מספר שדות כאלה, הרי זה בזבוז זיכרון לנצל 8 סיביות (char), מכיון שכל שדה כזה זקוק לסיבית אחת בלבד. כדי לפתור בעיה זו, מאפשרת השפה להגדיר **סיביות** בשדות ולהתייחס אליהם בהתאם. הצורה הכללית של הגדרת סיביות היא:

```
type member : nbits;
```

כאשר type מסמן את סוג המשתנה ויכול להיות סוג של שלם בלבד (char, uchar, int, uint, short, ushort, long, ulong), לאחר מכן מופיע שם השדה, ולאחריו מופיע הסימן ":" ואחריו מספר הסיביות שמשתתפות בשם הנוכחי. הדוגמה הבאה מראה מחלקה עם שדה המחולק לסיביות. במחלקה flags יש שלוש סיביות שמסודרות בשלם חסר סימן. המחלקה מאפשרת להכניס ערכים לסיביות השונות ולקבל את ערכם בעזרת פונקציות גישה.

```
class flags {
        unsigned keyword_bit : 1;
        unsigned extern_bit : 1;
        unsigned static_bit : 1;
public:
        flags()
        { keyword_bit = extern_bit = static_bit = 0; }
        void set_keyword_bit(unsigned v)
        { keyword_bit = v; }
        unsigned get_keyword_bit() const
        { return keyword_bit; }
        unsigned get_extern_bit() const
        { return extern_bit; }
};
```

הדוגמה הבאה מדגימה את השימוש בשתי מחלקות אלו. הפונקציה הראשית מגדירה מספר אובייקטים מסוג Union ומכניסה לתוכם ערכים. כמו כן, היא מגדירה אובייקט מסוג flag ונותנת ערכים לסיבית המסומנת כ-keyword.

```
#include <iostream.h>

int main()
{
    Union ui(1), uc('a'), uf(float(4.1));
    flags flgs;
    flgs.set_keyword_bit(1);
    cout << "\nsizeof(double) = " << sizeof(double)
         << "\nsizeof(ui) = " << sizeof(ui)
         << "\nui float val = " << ui
         << "\nuf float val = " << float(uf) << endl;

    cout << "\nsizeof(unsigned) = " << sizeof(unsigned)
         << "\nkeyword_bit = " << flgs.get_keyword_bit()
         << "\nextern_bit  = " << flgs.get_extern_bit()
         << endl;
    return 0;
}
```

הפלט של התוכנית הזו יהיה:

```
sizeof(double) = 8
sizeof(ui) = 10
ui float val = 1.53828e-29
uf float val = 4.1

sizeof(unsigned) = 2
keyword_bit = 1
extern_bit  = 0
```

הגודל של ui הוא עשרה בתים, לעומת שמונה בתים של double, מאחר והוא מחזיק שדה נוסף המתאר את סוג השדה הפעיל בכל רגע באיחוד. הגודל של אובייקט מסוג flag הוא שני בתים. אם היינו מחזיקים את שלושת השדות בתווים, היינו מקבלים שלושה בתים. בדרך זו חוסכים בית אחד.

## 3.13 מצביעים לשדות ופונקציות במחלקה

ב-++C קיימת אפשרות להגדיר **מצביעים לשדות** או **מצביעים לפונקציות מחלקה**. בסעיף זה נלמד כיצד ניתן להגדיר מצביעים מסוג זה. בסעיף זה נשתמש, לצורך ההדגמה, במחלקה הבאה:

```
class Pointers {
  int x1;
public:
  int x2;
```

```
  Pointers(int xo) : x1(xo), x2(xo) {}
  void set_x1(int x) { x1 = x;}
  void set_x2(int x) { x2 = x; }
  int get_x1() const { return x1; }
// A friend which outputs an object of Pointers to ostream.
   friend ostream &operator<<(ostream &out,const Pointers &p)
          { return (out << p.x1 << "\t" << p.x2 << endl);
}
};
```

למחלקה יש שני שדות שלמים, הראשון בחלק הפרטי והשני בחלק הציבורי. למחלקה יש גם שתי פונקציות שמאפשרות לעדכן את ערכי השדות האלה. לשדה הפרטי x1, יש פונקציית שירות (set_x1) המאפשרת לקבוע את ערכו. יש לו גם פונקציית שירות (get_x1) המאפשרת לקבל את ערכו.

האופרטור "<<" מוגדר למחלקה זו כ**חבר** (friend) של המחלקה. הסיבה שאופרטור זה הוא חבר, ולא פונקציה במחלקה, היא שפונקציה במחלקה חייבת לקבל אובייקט של המחלקה כאופרנד שמאלי. זהו אחד המקרים הבודדים שיש הצדקה לשימוש במנגנון החבר.

## 3.13.1 מצביעים לשדות

מצביע לשדה של מחלקה X כלשהי, שסוג השדה הוא **type**, מוגדר באופן הבא:

```
type X::*p;
```

מוסיפים את האופרטור רזולוציה (**::**) של המחלקה לפני סימן המצביע "*". האופרטור '::' נקרא **אופרטור רזולוציה** (resolution operator), או **אופרטור הבחנה**, משום שהוא מאפשר הבחנה בטווח של משתנה. למשל, הגדרת p היא מצביע לשדה של מחלקה x, ולא של מצביע גלובלי.

כאשר נבקש להגדיר מצביע לשדות השלמים של המחלקה הזו נעשה זאת כך:

```
// apply some member pointer manipulations
        int Pointers::*p;
        p = &Pointers::x2;
```

השימוש במצביע לשדה של מחלקה חייב לבוא בצמוד לאובייקט. הסיבה לכך היא שאין משמעות למצביע כזה ללא אובייקט מתאים. לכן, במקרה הכללי של מחלקה כלשהי X נשתמש במצביע בצורה הבאה:

```
X x, *xp;
xp = &x;
x.*p = val;
xp->*p = val;
```

זו הסיבה שהפעלה של מצביע לשדה של אובייקט נעשית דרך אובייקט נתון. האובייקט יכול להיות אובייקט, או מצביע לאובייקט. במקרה השני (מצביע לאובייקט) ההתייחסות למצביע לשדה היא דרך המצביע לאובייקט. בדוגמה שלנו המצביע לשדה (p) הוא מסוג שלם והשימוש בו הוא כזה:

```
Pointers x(0), *px;
cout << "\n after definition should be zero x = " << x;
x.*p = 2;
cout << "\n after assign 2 to x2 x = " << x;
px = &x;
px->*p = 3;
cout << "\n after assign to x2 value 3 x = " << x;
```

### 3.13.2 מצביעים לפונקציות של אובייקטים

ב-++C יש אפשרות להגדיר מצביעים לפונקציות של מחלקות. מצביעים אלה דומים בהגדרתם למצביעים רגילים, פרט לכך שהם משתמשים באובייקט שמסופק להם. באופן כללי עבור מחלקה X ופונקציה f נגדיר את המצביע באופן הבא:

```
ret-type (X::*func_ptr)(arg-list);
```

לכל מצביע לפונקציה יש לפרט את הערך שמחזירות פונקציות המוצבעות על ידו, ואת רשימת הפרמטרים שהוא מקבל. הפעלה של פונקציה כזו חייבת להיות דרך **אובייקט**. לדוגמה, כשיש מצביע כזה לאובייקט px, הפעלת פונקציה כזו תיעשה כך:

```
(px->*func_ptr)(arg-list);
```

אם נחזור לדוגמה הקודמת, של המחלקה Pointers, כאשר מוגדר מצביע לאובייקט, נגדיר מצביע לפונקציה של המחלקה. נאתחל את ערכו של המצביע לפונקציה כך שיצביע לפונקציה set_x1, ואז נקרא לפונקציה דרך המצביע עם ארגומנט 1. קריאה זו תכניס את הערך 1 לשדה x1.

```
void (Pointers::*funcptr)(int);
funcptr = Pointers::set_x1;
(px->*funcptr)(1);
cout << "\n after assign 1 to x1 x = " << x;
```

לסיכום נושא מצביעים לשדות של מחלקות, או לפונקציות של מחלקות, אני רוצה לציין שלשפת ++C יש מנגנונים רבי עוצמה אחרים. מנגנונים אלה, שנכיר בהמשך הספר, עדיפים על מצביעים לשדות או פונקציות מחלקה. השימוש במצביעים אלה הופך את הקוד לבלתי קריא, והייתי ממליץ להימנע משימוש בהם, אלא במקרים מיוחדים והם נדירים ביותר.

## 3.14 סיכום

בפרק זה למדנו את מושג המחלקה ב-++C. המחלקה ב-++C היא מודול שמסתיר את מבנה הנתונים אשר עליו הוא פועל, ומספק פונקציות שירות שמודולים אחרים יכולים לקרוא. פונקציות אלו מספקות אלגוריתמים שונים הפועלים על מבנה הנתונים של המחלקה.

בפרק זה למדנו להבדיל בין מחלקה לאובייקט. מחלקה היא תבנית שמאפיינת את כל האובייקטים במחלקה. אובייקט הוא מופע של מחלקה. משתנים הם אובייקטים.

כתוצאה מהסתרת המידע, והשימוש בפונקציות שירות לקבלת שירותים על מבנה הנתונים, אפשר לשנות את מבנה הנתונים באופן כזה שחלקי התוכנה האחרים, המשתמשים במחלקה זו, לא ישתנו. שינוי זה מאפשר ליצור תוכנה באיכות גבוהה החסינה לשינויים וניתנת לשימוש במספר רב של מקומות ללא צורך בשכתובה.

המחלקה ב-++C יכולה להגדיר אופרטורים בסיסיים שיפעלו על אובייקטים של המחלקה. המחלקה יכולה, למשל, להגדיר אופרטורים כגון "+", "=", "==" וכו'. הגדרה כזו מאפשרת לכתוב מחלקות שהאובייקטים שלהן מתנהגים כמו משתנים בסיסיים של השפה, וכך להרחיב את השפה בסוגים חדשים.

ראינו שאפשר להגדיר שדות, או פונקציות סטטיות אשר משותפות לכל האובייקטים במחלקה. שדות סטטים שימושים למקרים בהם רוצים לשמור מידע המשותף לכל האובייקטים במחלקה. פונקציות סטטיות אינן קשורות לאובייקט מסוים, אך יש להן גישה לחלק הפרטי של אובייקט נתון.

לכל מחלקה יש אזורי גישה: public, private, שקובעים את הגישה לשדות, או לפונקציות של המחלקה מקטעים אחרים של התוכנה. public מסמן גישה ציבורית, ואילו private מסמן גישה פרטית המותרת רק לפונקציות של המחלקה, או לחברים של המחלקה.

אפשר להגדיר מחלקות מקוננות, אשר מוגדרות בתוך גוף הקוד של מחלקה אחרת. **מחלקה מקוננת** (nesting function) יכולה להיות מוגדרת בכל אחד מאזורי הגישה של המחלקה העוטפת. אזור הגישה קובע את הגישה למחלקה המקוננת.

## 3.15 שאלות

1. ממש את האופרטורים ">", "<=", ">=" במחלקה String.
2. ממש את המחלקה String בעזרת malloc, free ו-realloc.
3. ממש את המחלקה String בעזרת האופרטורים new ו-delete. דאג שהקצאת זיכרון חדשה לא תתרחש בכל שינוי המחרוזת שהאובייקט מצביע עליה.
רמז: הקצה זיכרון גדול למהדר ורק כשנגמר הזיכרון הקצה זיכרון אחר.
4. האם צריך לכתוב אופרטור מסוג String &operator+=(const char *) ? נמק!

5. מה קורה כאשר מעבירים מצביע שהוא NULL לבנאי של String?

6. הגדר אופרטור העתקה למחלקה Person.

7. כתוב מחלקה שמייצגת קובץ במערכת, ומגבילה את מספר הקבצים הפתוחים של התוכנית למספר נתון, שאינו קבוע אלא ניתן לשינוי.

8. כתוב מחלקה אשר מייצגת ספרייה (directory) במערכת ההפעלה, ומגבילה את מספר הקבצים בה למספר נתון. מה ההבדל בין מחלקה זו למחלקה מהשאלה הקודמת?

# פרק 4

# קלט ופלט

בפרק זה נראה את אמצעי הקלט/פלט של C++. אמצעים אלה אינם חלק מהשפה אבל הם בנויים בתקן שלה. אפשר להשתמש בספרייה הסטנדרטית של C גם ב-C++ אבל הספרייה של C++ עדיפה. ספריית הפלט/קלט מאפשרת ל**העמיס אופרטורים** כגון ״>>״ ו-״<<״ לאובייקטים, כדי לשלוח את האובייקטים ישירות לפלט, או לקרוא אובייקט שלם מהקלט בדרך זו.

בנוסף, ספרייה זו בטוחה יותר לשימוש. עובדה זו נובעת מהיכולת של C++ להעמיס אופרטורים לפי סוגי המשתנים. בפרק זה נראה מדוע ספרייה זו עדיפה.

## 4.1 חסרונות הספרייה הסטנדרטית של C

מדוע עלינו להשתמש בספרייה חדשה? האם הספרייה של C, לה אנו רגילים זמן רב, אינה טובה? בסעיף זה נענה על שאלות אלו, ונעמוד על היתרונות והחסרונות של ספריית הקלט/פלט הסטנדרטית של C. כאשר רוצים לפתוח קובץ בספרייה זו יש לבצע קריאה, למשל :

```
FILE *fp = fopen("file", "wr");
```

הפונקציה **fopen** פותחת את הקובץ file. הקובץ נפתח לקריאה ולכתיבה, לפי הארגומנט השני. הפונקציה מחזירה **ידית** (handle) לקובץ הפתוח. הפונקציות האחרות של הספרייה משתמשות בידית האחרונה כדי לבצע את הפעילויות שלהן. ספריית הקלט/פלט של C היא מודול המטפל בקריאה, או כתיבה לקבצים. מודול זה דומה למודולים עליהם למדנו בפרק 2.

נניח שיש לנו פונקציה שפותחת קובץ כלשהו ומעבדת אותו :

```
void f(const char *name)
{
      FILE *fp = fopen(name, "r");
      // .. do something
}
```

לפונקציה הזו יש בעיה. המתכנת שכח לסגור את הקובץ. התוכנית תמשיך לרוץ, אבל לאחר הפעלה של הפונקציה f מספר פעמים תפסיק התוכנית לתפקד, מכיון שהיא אינה סוגרת את הקבצים ואינה מחזירה משאבים למערכת. מסקנה: החיסרון הראשון של מודול זה הוא שיש לזכור לסגור קבצים (חיסרון דומה ראינו בפרק 2).

נניח שמתכנת רוצה לכתוב ערך כלשהו לתוך הקובץ. לשם כך הוא ישתמש בפונקציה fprintf באופן הבא:

```
void f(const char *file)
{
      FILE *fp = fopen(file, "w");
      int x;
      fprintf(fp, "%s", x);
}
```

הפונקציה **fprintf** מקבלת, כארגומנט ראשון, ידית (handle) לקובץ שנפתח בעזרת fopen. הארגומנט השני של הפונקציה הוא מחרוזת המתארת כיצד יש להתייחס לארגומנטים המופיעים בהמשך. במקרה זה המחרוזת מסמנת שיש להדפיס מחרוזת של תווים. הבעיה כאן היא שהארגומנט שניתן לפונקציה הוא שלם, כך שאין אפשרות לדעת מה יודפס לקובץ. התנהגות התוכנית אינה צפויה. ההצהרה של הפונקציה fprintf היא כדלהלן:

```
void fprintf(FILE *, const char *format, ...);
```

חוסר התיאום בין המחרוזת לבין הארגומנטים הבאים אחריה הוא המקור לבעיות. בעיות אלו אינן יכולות להתגלות ברמת הידור התוכנית, מכיון שהארגומנט השלישי והלאה של הפונקציה fprintf הוא "..." (הפונקציה מקבלת מספר לא מוגדר של ארגומנטים), לכן המהדר אינו יכול לבדוק את תקינות הפרמטרים.

אותן בעיות יכולות להיווצר כתוצאה מהפעלת הפונקציות **fscanf** הקוראות קלט מקובץ ועורכות אותו (פורמט). במקרה הטוב, תופסק התוכנית בנקודת הקריאה לפונקציה. במקרה הרע, נגלה את הבעיות מאוחר יותר.

בעיה אחרת יכולה להיגרם אם מתכנת משתמש בידית לקובץ לפני שהוא פתח את הקובץ עצמו. אין דרך למנוע מהמתכנת לעשות פעולה כזו הגורמת לשגיאה בזמן ריצה.

```
void f(...)
{
      FILE *fp;
      fprintf(fp, "%s", "A string");
}
```

התנהגות הפונקציה אינה מוגדרת. במקרה הטוב התוכנית תפסיק בנקודת הקריאה לפונקציה fprintf, במקרה הפחות טוב, התוכנית תמשיך עם תוצאות בלתי צפויות במקום אחר ללא קשר למקום בו נעשתה השגיאה. נרצה דרך אלגנטית ופשוטה, בה יכול המהדר לגלות בעיות בזמן ההידור ולהתריע עליהן (דבר שהיה חוסך זמן המבוזבז לגילוי ואיתור בעיות כאלו).

## 4.2 קלט/פלט פשוט ב-++C

ב-++C יש מספר מחלקות אשר תומכות בקלט ובפלט. הן תומכות בקלט, או בפלט, של כל סוגי הנתונים התקניים בשפה. סוגי נתונים אחרים, כמו אובייקטים שהמתכנת מגדיר, אינם נתמכים ישירות על ידי ספריית הקלט/פלט. אולם, מתכנת המגדיר אובייקטים אלה יכול להגדיר להם אופרטורים מתאימים המאפשרים לכתוב, או לקרוא, את האובייקטים כאילו היו אובייקטים תקניים של ++C.

**קלט/פלט בסיסי** מתבצע על ידי האובייקטים הבסיסיים: "<<" ו-">>". האופרטור ">>" מוגדר עבור קלט, והאופרטור "<<" מוגדר עבור פלט. אופרטורים אלה עורכים מחדש את הנתונים (reformat), כלומר, לוקחים מחרוזת וממירים אותה למשתנה המתאים.

### 4.2.1 העמסת אופרטורים

יש מספר אופרטורים מסוג "<<" ומסוג ">>", עבור כל אחד מסוגי הנתונים הבסיסיים של השפה. אפשרות זו נקראת **העמסה** (overloading). המהדר בוחר את הגירסה המתאימה של האופרטור לפי סוג המשתנה. לדוגמה, נתונה מחלקה X ולה מספר אופרטורים "<<" אשר כתובים במשפטים שונים:

```
class X {
// ...
public:
      X &operator<<(int v);      // opr 1
      X &operator<<(char);       // opr 2
      X &operator<<(char *);     // opr 3
};

X x;
int i;
char c, *str;
...
x << i << c << str;
```

בקטע הקוד הזה יש שלושה אופרטורים. הראשון מקבל ערך שלם, השני מקבל תו והשלישי מקבל מצביע לתו. אפשר להפעיל את האופרטורים בשרשרת, מכיון שכל אחד מהם מחזיר ייחוס (reference) לאובייקט מסוג X. המהדר בוחר את האופרטור המתאים על פי סוג הארגומנט המועבר לאופרטור. עבור המשתנה i ייבחר האופרטור הראשון (opr 1), עבור המשתנה c יופעל האופרטור השני (opr 2) שפועל על תווים, ואילו עבור המצביע לתווים יופעל האופרטור השלישי, שפועל על מצביעים לתווים.

## 4.2.2 אובייקטים בסיסיים

האובייקטים הבאים קיימים בכל תוכנית המשתמשת בספריית הפלט/קלט הסטנדרטית של C++ :

- cin שייך למחלקה istream ונועד לקלוט נתונים מהקלט הסטנדרטי.
- cout שייך למחלקה ostream ומאפשר פלט של נתונים לפלט הסטנדרטי.
- cerr שייך למחלקה ostream ומאפשר פלט של נתונים לאמצעי דיווח השגיאות הסטנדרטי.
- clog שייך למחלקה ostream ומאפשר לכתוב נתונים לאמצעי הרישום הסטנדרטי.

אובייקטים אלה קשורים, בדרך כלל, למסך או לחלון שממנו מופעלת התוכנית. כדי להשתמש בהם יש לכלול בתוכנית את הקובץ **iostream.h**. הדוגמה הבאה מדגימה שימוש פשוט באובייקטים אלה.

```
#include <iostream.h>
int main()
{
	char c;
	cout << "\nHello enter any key:" ;
	cin  >> c;
	cout << "The key you typed in is:" << c << "\n";
	return 0;
}
```

התוכנית מדפיסה לפלט את המחרוזת: "Hello enter any key:". לאחר ההדפסה לפלט ממתינה התוכנית לקלט כלשהו מ**הקלט הסטנדרטי** (standard input). הקלט מוכנס למשתנה c. לבסוף, מדפיסה התוכנית את הקלט לפלט ומסיימת.

אפשר לראות איך בפעולה אחת ניתן לשלוח לפלט אחד מספר משתנים. באופן דומה ניתן לקרוא מספר משתנים בפקודה אחת. למשל:

```
struct point {
	int x,y;
};

int main()
{
	point pnt;
	cout << "\nenter x and y:"
	cin >> pnt.x >> pnt.y;
	return 0;
}
```

תוכנית זו קולטת שני ערכים שלמים מהמשתמש ומכניסה אותם למשתנים x ו-y של הרשומה pnt. על המשתמש להכניס רווח בין שני הערכים (כדי שהתוכנית תקבל שני ערכים). אם המשתמש אינו מכניס רווח בין שני המספרים תזהה הספרייה מספר אחד בלבד ותמתין למספר נוסף.

אם המשתמש מכניס ערך שאינו שלם, יעבור האובייקט cin למצב לא תקין. כדי לנקות מצב לא תקין זה, צריך המתכנת לקרוא לפונקציה clear שתנקה את הדגלים המסמנים את מצב האובייקט.

## 4.3 קריאה מקבצים

עד עתה ראינו קריאה וכתיבה (קלט/פלט) לאמצעים הסטנדרטיים של התוכנית. אפשר גם לקרוא ולכתוב מקבצים. בסעיף זה נלמד כיצד קוראים מקבצים. המחלקה שמבצעת קריאה מקבצים היא **ifstream** וכדי להשתמש בה, יש לכלול את הקובץ fstream.h בקובץ המתאים.

### 4.3.1 קריאה עם עריכה

קריאה עם עריכה (על פי פורמט) היא קריאה של מחרוזת תווי ASCII והמרתה לסוג המשתנה המבוקש. למשל, אם אנו מנסים לקרוא שלם, תתבצע המרה אוטומטית של המחרוזת לשלם.

המחלקה שמטפלת בקריאה מקבצים היא **ifstream**. הבנאי של המחלקה יכול לקבל את שם הקובץ שאליו יהיה קשור האובייקט ועליו יתבצעו פעולות הפלט.

פעולת פתיחת קובץ יכולה להתרחש גם תוך שימוש בפונקציה open המקבלת את שם הקובץ ופותחת אותו. לאחר פתיחת הקובץ אפשר לבדוק, בכל רגע, שהאובייקט נמצא במצב תקין (למשל, שקובץ כזה באמת קיים) על ידי קריאה לפונקציה good שמחזירה ערך השונה מאפס אם הקובץ נמצא תקין, וערך אפס אם לא.

לחילופין, אפשר לבדוק את האובייקט בביטוי לוגי שיציג ערך אמת (true) אם האובייקט תקין, ואחרת - ערך שקר (false). דבר זה ניתן לביצוע, כי המחלקה מממשת אופרטור המרה של אובייקט המחלקה אל מצביע השונה מאפס כאשר האובייקט במצב תקין, ואחרת לאפס.

הפונקציה הבאה, readFile, מקבלת שם קובץ לקריאה וקוראת אותו. היא מגדירה אובייקט file ששייך למחלקה ifstream ומקבל את שם הקובץ. לאחר מכן, מוכנסת המחרוזת הראשונה מהקובץ לתוך מערך התווים buf. בתוך לולאת while נערכת בדיקה אם הקובץ תקין. הבדיקה המתבצעת היא, האם הוא פתוח ועדיין לא הגענו לסופו (EOF). כל עוד הקובץ נמצא במצב תקין נקראות ממנו מחרוזות ונכתבות לפלט הסטנדרטי.

בסופו של דבר מחזירה הפונקציה את מספר המילים שנקראו מהקובץ. אובייקט מסוג ifstream סוגר את הקובץ שאליו הוא קשור בפונקציית הפירוק שלו. מכיון שפונקציית הפירוק של האובייקט נקראת באופן אוטומטי כאשר יוצאים מהפונקציה, היא גם תסגור את הקובץ באופן אוטומטי. לכן, אין צורך לסגור את הקובץ באופן ידני בסיום הפונקציה.

כזכור, במקרה של מודולים מצב זה אינו נכון. אם לא היינו סוגרים את הקובץ היינו מבזבזים את משאבי המערכת, ובשלב מאוחר יותר היתה המערכת מפסיקה לתפקד.

למרות האמור, אפשר להשתמש בפונקציה close כדי לסגור את הקובץ באופן מפורש. פונקציה זו קיימת גם עבור אובייקטי קלט וגם עבור אובייקטי פלט. הפונקציה אינה מקבלת ארגומנטים, או מחזירה ערך.

```
int readFile(const char *fileName)
{
	const int bufSize = 128;
	int count = 0;
	char buf[bufSize];
	ifstream file(fileName);
	cout << endl;
	file >> buf;
	while (file) {
		cout << buf;
		file >> buf;
		count++;
	}
	return count;
}
```

אם היינו יודעים את מבנה הקובץ, למשל שני שלמים ולאחריהם מחרוזת, היינו יכולים לקרוא זאת כך:

```
int i1, i2;
char buf[64];
file >> i1 >> i2 >> buf;
```

העריכה היתה מתבצעת באופן אוטומטי על ידי האובייקט הקורא (file). בשתי ההפעלות הראשונות של האופרטור "<<" היתה מתבצעת עריכה לשלמים והערכים המתאימים היו מוכנסים למשתנים i1 ו-i2, והקריאה השלישית היתה מכניסה ערכים מתאימים למחרוזת.

## 4.3.2 קריאה ללא עריכה

**קריאה ללא עריכה** (unformatted input) מיועדת לקריאת קבצים במבנה בינארי. קבצים כאלה נחשבים, על ידי האובייקט הקורא כזרם של בתים. האובייקט הקורא (אובייקט הקלט) אינו מבצע כל המרה על בתים אלה. קריאות של קבצים בינארים

מתבצעות על ידי הפונקציה read של המחלקה ifstream. הפונקציה מקבלת כארגומנט ראשון את החוצץ שלתוכו יש לקרוא את זרם הבתים. הארגומנט השני שמועבר לפונקציה הוא מספר הבתים שיש לקרוא. הפונקציה מחזירה ייחוס לאובייקט הנוכחי, וכך אפשר לבדוק את תקינות הקריאה, או לבצע קריאה בשרשרת.

הפונקציה הבאה מבצעת קריאה בינארית של קובץ, ומניחה שבקובץ מצויים מבנים מסוג point. אם בקובץ יהיו מבנים מסוג אחר, תהיינה תוצאות הקריאה בלתי צפויות. הפונקציה מגדירה אובייקט מסוג ifstream שמשמש לקריאת הקובץ. האובייקט מקבל את שם הקובץ כפרמטר ראשון; הפרמטר השני מסמן שהקובץ הוא בינארי. דבר זה ייחודי למערכת ההפעלה DOS שמבדילה בין קבצים בינאריים לאחרים. פרמטר זה אינו קיים בתקן הספרייה, ובוודאי שאינו נמצא בספריות שבמערכת ההפעלה UNIX.

לאחר מכן, הפונקציה קוראת בלולאה מבנה נוסף מהקובץ ומדפיסה את המבנה לפלט הסטנדרטי. בתוך הלולאה מוגדל מונה המציין את מספר הרשומות שנקראו מהקובץ. בסוף הקריאה מוחזר ערך המונה.

```
int readBinFile(const char *fileName)
{
    point pnt;
    ifstream file(fileName, ios::binary);
    int count = 0;
    cout << "\nreadBinFile" << endl;   // (1)
    while (file && file.read((char*)&pnt, sizeof(pnt))) {
        cout << "point.x = " << pnt.x
              << " ,point.y=" << pnt.y << endl;
        count++;
    }
    return count;
}
```

השורה בקוד זה המסומנת ב-(1) מדפיסה את שם הפונקציה ואחריו endl. הדבר מסמן לאובייקט cout להתחיל בשורה חדשה. בנוסף, הוא מוודא שכל מה שנמצא בחוצצים הפנימיים של האובייקט cout ושל המערכת ברגע זה, יישלח לתצוגה על המסך. תו זה נקרא **מְטַפֵּל** (manipulator - נחזור לדון בו בהמשך פרק זה).

## 4.4 כתיבה לקבצים

גם בכתיבה לקבצים יש שתי אפשרויות: **כתיבה עם עריכה** (formatted write) **וכתיבה בינארית** (binary write). **בכתיבה עם עריכה** מומרים הערכים של הפרמטרים למחרוזת ASCII כדי שעין אנוש תוכל לקרוא את הקובץ לאחר מכן. **בכתיבה בינארית** לקבצים לא מתבצעות המרות והמשתנים השונים נכתבים כזרם של בתים, ולמעשה - סיביות.

המחלקה שמאפשרת כתיבה לקבצים היא ofstream. כדי להשתמש במחלקה זו יש לכלול את הקובץ fstream.h בקבצים המתאימים שמשתמשים באובייקטי פלט.

## 4.4.1 כתיבה עם עריכה

בכתיבה עם עריכה מומר הערך המבוקש למחרוזת קודם לפעולת הכתיבה ורק אז נשלחת המחרוזת לקובץ. כדי להשיג מטרה זו מנצלת הספרייה את תכונת ההעמסה (overloading) של C++. במקרה זה נעשית ההעמסה של האופרטור "<<" (אופרטור הפלט). ההעמסה של האופרטור נעשית באופן דומה לאופרטור הקודם. כל אופרטור כזה מחזיר ייחוס לאובייקט הנוכחי (שעליו פועל האופרטור), כדי לאפשר הפעלת האופרטור בשרשרת.

הפונקציה הבאה, writeFile, שולחת מחרוזות לקובץ נתון. היא מקבלת כפרמטר את שם הקובץ שאליו יש לכתוב את המחרוזות, מגדירה אובייקט מסוג ofstream ומעבירה אליו את שם הקובץ לכתיבה. פעולה זו פותחת את הקובץ לכתיבה. אפשר להשיג את אותו האפקט על ידי קריאה לפונקציה open של אובייקט זה.

לאחר מכן, הפונקציה קוראת מחרוזות מהקלט הסטנדרטי, כל עוד המחרוזת שונה מ-end. דבר זה מתבצע בלולאת while. בגוף הלולאה נשלחת המחרוזת לקובץ הפלט ונקראת מחרוזת נוספת. תוך כדי כך גם מוגדל מספר המחרוזות שנכתבו לקובץ. לבסוף מוחזר מונה מספר המחרוזות (count).

```
int writeFile(const char *fileName)
{
    const int bufSize = 128;
    char buf[bufSize];
    int count = 0;
    ofstream file(fileName);
    cin >> buf;
    while (strcmp("end", buf)) {
       file << buf;
       cin >> buf;
       count++;
    }
    return count;
}
```

הפונקציה יכולה לכתוב כל משתנה בסיסי לקובץ הקלט תוך שימוש באופרטור "<<". האופרטור מועמס גם הוא עבור כל המשתנים הבסיסיים.

## 4.4.2 כתיבה בינארית לקבצים

בכתיבה בינארית לקבצים לא מתרחשת המרה של זרם הבתים. החוצצים המועברים לפונקציית הכתיבה נכתבים ישירות לקבצים כמו שהם. הפונקציה שמשמשת לכתיבה בינארית היא write, המקבלת את כתובת ההתחלה של החוצץ אותו יש לכתוב לקובץ. בנוסף, הפונקציה מקבלת את גודל החוצץ, או את מספר הבתים שיש לכתוב לקובץ.

הפונקציה מחזירה ייחוס לאובייקט שעליו היא פועלת, כדי לאפשר שרשור של קריאות לפונקציה זו.

הפונקציה writeBinFile כותבת שלוש רשומות מסוג point לקובץ נתון. הפונקציה מקבלת את שם הקובץ כפרמטר. היא מגדירה אובייקט מסוג ofstream ומעבירה לו את שם הקובץ. כמו במקרה של קריאה בינארית, מעבירה הפונקציה **מוד בינארי** (binary mode) לבנאי של האובייקט file. דבר זה אינו חלק מהסטנדרט והוא בר-יישום רק במקרה הפרטי של מערכת ההפעלה DOS, שמבדילה בין קבצים בפורמט ASCII לבין קבצים בפורמט בינארי. במערכות הפעלה אחרות אין צורך להבדיל בין סוגי קבצים.

הפונקציה כותבת שלוש רשומות לקובץ, כשהשדות x ו-y של כל רשומות מקבלים את הערך הנוכחי של המשתנה **I** ובמקרה זה, את הערכים 0, 1 ו-2. לבסוף, מחזירה הפונקציה את מספר הרשומות שנכתבו לקובץ (cnt).

```
int writeBinFile(const char *fileName)
{
    const int cnt = 3;
    point pnt;
    ofstream file(fileName, ios::binary);
    for (int i=0; i<cnt; i++) {
       pnt.x = pnt.y = i;
       file.write((char*)&pnt, sizeof(pnt));
    }
    return cnt;
}
```

## 4.5 שילוב קלט ופלט

לעיתים עלינו גם לקרוא וגם לכתוב לקבצים. אפשר להגדיר אובייקט שמבצע את הקריאה ואובייקט שמבצע את הכתיבה. אך ניתן גם להגדיר אובייקט בעל יכולת כפולה, שיבצע הן קריאה והן כתיבה. בסעיף זה נראה את שתי האפשרויות האלו.

### 4.5.1 שילוב בין אובייקטי קלט לפלט

כאשר יש פעולות קריאה מקובץ אחד ופעולות כתיבה לקובץ שני, אפשר לבצע זאת על ידי אובייקטים נפרדים. אובייקט אחד אחראי לקריאה מהקובץ האחד, והאובייקט השני אחראי לכתיבה לקובץ השני.

הפונקציה copyFile מעתיקה קובץ מקור לקובץ יעד. הפונקציה מקבלת, כפרמטר ראשון, את שם קובץ המקור וכפרמטר שני את שם קובץ היעד. הפונקציה מגדירה משתנה מסוג שלם (int), שישמש לקריאת התווים בקובץ המקור. הסיבה להגדרת משתנה מסוג שלם כשצריך לקרוא תו תתברר בהמשך.

הפונקציה מגדירה שני אובייקטים:

- out - אובייקט מסוג **ofstream**, המשמש לכתיבה לקבצים.
- in - אובייקט מסוג **ifstream**, המשמש לקריאה מקבצים.

בהמשך, הפונקציה קוראת בלולאה לפונקציות של אובייקטי הקלט והפלט שמבצעות קריאה וכתיבה של תו אחד בלבד. הפונקציה get קוראת תו אחד מהאובייקט in, ומחזירה את ערכו כשלם. הפונקציה מחזירה את הערך EOF כאשר מגיעים לסוף הקובץ. הערך EOF מוגדר כמינוס אחד, לכן אי אפשר להגדיר את המשתנה c כתו חסר סימן, לפיכך c מוגדר כשלם. כל עוד נקרא תו השונה מסוף קובץ (EOF) קוראת הפונקציה לפונקציה put של האובייקט out, השולחת תו אחד לאובייקט out.

```
void copyFile(const char *dst, const char *src)
{
    int c;
    ofstream out(dst);
    ifstream in(src);
    while ((c = in.get()) != EOF)
      out.put(char(c));
}
```

בכל שלב של הלולאה נקרא תו אחד מקובץ הקלט ונכתב תו אחד לקובץ הפלט. הדבר גורר קריאה לפונקציה כדי לקרוא כל תו בקלט, ועוד קריאה לפונקציה כדי לכתוב את התו. אפשר היה לשפר מצב זה על ידי קריאה לפונקציה read. במקרה זה הינו קוראים מספר גדול יותר של תווים, 1024 למשל.

ומדוע שלא נשתמש בפונקציה read? מכיון שהיא אינה מחזירה את מספר הבתים שאכן נקראו. כדי לקבל את המספר האמיתי של בתים שנקראו, ניתן להשתמש בפונקציה gcount, המחזירה את מספר הבתים שנקראו בפעולת הקריאה האחרונה. אני משאיר לקורא בעיה זו כתרגיל.

## 4.5.2 שימוש באובייקטי קריאה וכתיבה

לעיתים, צריך לקרוא ולכתוב לאותו קובץ, כמו בעבודה בדיסקים מגנטיים. בספריית הקלט/פלט הסטנדרטי יש אובייקטים שיכולים לקרוא ולכתוב בעת ובעונה אחת לאותו קובץ. אובייקטים מסוג fstream מאפשרים קריאה, או כתיבה של אותו קובץ בו-זמנית. כדי להשתמש באובייקטים אלה יש לכלול את הקובץ fstream.h בתוכנית.

## 4.5.3 קביעת מיקום מצביע הקריאה או הכתיבה

לאובייקטים פלט/קלט כלשהם יש אפשרות לקבוע את המיקום בקובץ שממנו תתבצע פעולת הקריאה, או הכתיבה הבאים. הדבר נכון לכל האובייקטים הקשורים לקבצים, ובפרט לאלה המסוגלים רק לקרוא, רק לכתוב או לקרוא ולכתוב.

כדי לקבוע את מיקום פעולת הקריאה הבאה, יש להשתמש בפונקציה seekg, כאשר g מסמל את המילה get. כדי לקבל את המיקום של פעולת הקריאה הבאה יש להשתמש בפונקציה tellg שמחזירה מספר מסוג שלם מורחב (long).

כדי לקבוע את מיקום פעולת הכתיבה הבאה יש להשתמש בפונקציה seekp, שקובעת את מיקום מצביע הכתיבה בקובץ. הפונקציה tellp מחזירה את ההיסט של מצביע הכתיבה מתחילת הקובץ. גם פונקציה זו מחזירה ערך של שלם מורחב.

הפונקציה changeOrderOfPoints מקבלת שם של קובץ שבו יש שלוש רשומות מסוג point ומחליפה את הסדר בין הרשומות. הרשומה הראשונה תעבור להיות הרשומה האחרונה, והרשומה האחרונה תהיה הראשונה.

```
void changeOrderOfPoints(const char *name)
{
  fstream file(name, ios::binary);
  file.open(name, ios::binary | ios::in | ios::out);
  if (!file)
      cout << "\nfailed to open the file" << endl;
      point p1, p2;
      file.seekg(sizeof(p1) * 2);
      file.read((char*)&p1, sizeof(p1));
      file.seekg(0);
      file.read((char*)&p2, sizeof(p2));
      file.seekp(sizeof(p2)*2);
      file.write((char*)&p2, sizeof(p2));
      file.seekp(0);
      file.write((char*)&p1, sizeof(p1));
      cout << "\nchange\n" << "p1.x=" << p1.x << ",
                  p1.y=" << p1.y<< "\np2.x=" << p2.x << ",
                  p2.y=" << p2.y << endl;
}
```

הפונקציה מקבלת את שם הקובץ עליו היא פועלת. הפונקציה מגדירה אובייקט מסוג fstream שמשמש לקריאה ולכתיבה של הקובץ הנתון. היא מעבירה לבנאי של האובייקט את שם הקובץ שעליו יש לבצע את פעולות הקריאה והכתיבה. לאחר מכן, היא משתמשת בפונקציה open, שלה מועבר שם הקובץ ואופן (מוד, mode) הפעולה על הקובץ. הקובץ נפתח במוד בינארי (ייחודי למערכת ההפעלה DOS ואינו כלול בסטנדרט של ספרייה זו) לקריאה ולכתיבה. כדי להשיג את הפונקציונליות הדרושה נבצע or על הסיביות המתאימות, ונעביר אותן לפונקציה open כפרמטר שני.

לאחר מכן, נבדק האובייקט file כדי לדעת אם הצליחה פעולת הפתיחה. אם לא - מודפסת הודעת שגיאה לפלט הסטנדרטי. הפונקציה מגדירה שתי רשומות מסוג point המשמשות להחלפת הרשומות בקובץ המתאים.

הפונקציה מבצעת הזזה של מצביע הקריאה בקובץ לתחילת הרשומה השלישית, ולאחר מכן היא קוראת את הרשומה הראשונה. בהמשך, מזיזה הפונקציה את מצביע הקריאה לתחילת הקובץ וקוראת את הרשומה השנייה.

בשלב הכתיבה מזיזה הפונקציה את מצביע הקריאה לרשומה השלישית וכותבת את הרשומה השנייה. מייד לאחר מכן, שוב מוזז מצביע הכתיבה והפעם לתחילת הקובץ. כעת נרשמת הרשומה הראשונה שנקראה. התוצאה של פעולה זו היא החלפת הערכים של הרשומה הראשונה והשלישית בקובץ.

## 4.6 עריכת נתונים בזיכרון

אפשר לערוך נתונים בזיכרון. אפשרות זו דומה למשפחת הפונקציות הקיימות ב-stdio.h : scanf. לצורך עריכת נתונים בזיכרון מוגדרות שלוש מחלקות :

- istrsteam - מאפשרת קלט נתונים ערוכים ממחרוזות.
- ostrstream - מאפשרת פלט נתונים בצורה ערוכה אל חוצץ בזיכרון.
- strstream - מאפשרת פלט וקלט של נתונים ערוכים.

כדי להשתמש במחלקות אלו עליך לכלול בתוכנית את הקובץ strstream.h.

הפונקציה strInputOutput מדגימה את השימוש בשתי המחלקות הראשונות. תחילה היא מגדירה אובייקט מסוג ostrstream ומבקשת מהמשתמש להקליד שני שלמים ומחרוזת. ערכים אלה נקראים לתוך המשתנים x1, x2 ו-buf, בהתאמה. לבסוף, מוכנס התו ends המסמן את סוף המחרוזת.

הפונקציה מגדירה אובייקט מסוג istrstream שמקבל את אזור הזיכרון שנצבר במהלך הפעולות הקודמות על ידי האובייקט str. הבנאי של istr מקבל גם את גודל אזור הזיכרון. לאחר מכן, הערכים שנכתבו קודם לכן מחולצים אל המשתנים x, y ו- buf1 שבזיכרון. לבסוף, משחררת הפונקציה את אזור הזיכרון שהוקצה על ידי האובייקט ostr.

הפונקציה str של המחלקה ostrstream מחזירה מצביע לאזור הזיכרון הפנימי שלתוכו נכתבים המשתנים בצורה ערוכה. אם משתמשים בפונקציה זו, האובייקט ostr **אינו** משחרר את אזור הזיכרון שהוא הקצה, לכן האחריות לשחרור עוברת לפונקציה שקראה לפונקציה זו.

```
void strInputOutput()
{
    ostrstream ostr;
    int x1, x2, x, y;
    char buf[32], buf1[32];
    cout << "enter two ints and a string:" ;
    cin >> x1 >> x2 >> buf;
    ostr << x1 << " " << x2 << " " << buf << ends;
    cout << ostr.str() << endl;
```

```
    istrstream istr(ostr.str(), strlen(ostr.str()) + 1);
    istr >> x >> y >> buf1;
    cout << "\nx=" << x << ", y=" << y << ", buf=" << buf;
    delete [] ostr.str();
}
```

אפשר להעביר לבנאי של ostrstream מצביע לאזור זיכרון ואת גודל אזור הזיכרון הנתון, אך אז האובייקט לא יקצה זיכרון בעת הצורך, אלא ישתמש אך ורק בחוצץ הנתון לו. במקרה כזה אין צורך לשחרר את החוצץ.

כמו לאובייקטי הקלט/פלט הקודמים, אפשר לשנות את מיקום הקריאה, או הכתיבה הבא תוך שימוש בפונקציות seekg ו-seekp. אפשר גם לקבל את המצביע הנוכחי של הקריאה, או הכתיבה, בעזרת הפונקציות tellp ו-tellg, בהתאמה.

האובייקט strstream משמש לעריכת נתונים המוזנים לזיכרון או שיוצאים ממנו. כלומר, קריאה או כתיבה לזיכרון בו-זמנית. אובייקט זה דומה לאובייקט המקביל, הפועל על קבצים, fstream.

## 4.7 שילוב פונקציות פלט/קלט

הפונקציה main משתמשת בכל הפונקציות שראינו עד עתה. היא מקבלת שם קובץ ומייצרת קובץ פלט ששמו streams.out. אם לא נמסור לתוכנית שם קובץ, היא תשתמש בקובץ foo.dat. בקובץ זה הרשומות יהיו בסדר הפוך!

```
int main(int argc, char *argv[])
{
    const char *fileName = argv[1];
    const char *outFile = "streams.out";
    if (argc != 2) {
        cerr << "usage: " << argv[0] << " file-name" << endl;
        cerr << "using file foo.dat" << endl;
        fileName = "foo.dat";
    }

    writeFile(fileName);
    readFile(fileName);
    writeBinFile(fileName);
    readBinFile(fileName);
    copyFile(outFile, fileName);
    changeOrderOfPoints(outFile);
    readBinFile(outFile);
    strInputOutput();
    return 0;
}
```

## 4.8 מניפולטורים

מניפולטורים (מְטַפְּלְלִים - Manipulators) הם פונקציות הפועלות על אובייקט מסוים ומשנות את מצבו, או קוראות לפונקציה של האובייקט ומחזירות ייחוס לאובייקט הנתון. הדבר מאפשר להפעיל את המניפולטורים ברצף הפקודות, ללא צורך בפתיחת משפט חדש.

ראינו מספר מניפולטורים בפרק זה: endl ו-ends. הראשון מכניס תו בקרה 'סוף שורה' לאובייקטים שמבצעים פלט לקבצים, והשני מכניס תו בקרה 'סוף מחרוזת' לאובייקטים מסוג ostrstream.

### 4.8.1 תמיכה במניפולטורים במחלקה נתונה

עבור מחלקה כלשהי X, המניפולטור (מְטַפְּלֵל) של המחלקה היא פונקציה f הנכתבת כך:

```
X &f(X&);
```

כדי שמחלקה X כלשהי תאפשר קריאה למניפולטורים כאלה, יש להגדירה כך:

```
class X {
//...
public:
//...
    void doIt();
    X &operator<<(int v) { ... }
    X &operator<<(X &(*f)(X&))
    { return (f(*this)); }
};
```

אם למחלקה יש אופרטורים מסוג "<<" נגדיר אופרטור נוסף מסוג זה. הוא יקבל מצביע לפונקציה שמחזירה ייחוס אל האובייקט שעליו היא פועלת. האופרטור יפעיל את הפונקציה ויחזיר את האובייקט שמחזירה הפונקציה.

כעת אפשר להגדיר מניפולטור כלשהו, או פונקציה:

```
X &f(X &x)
{
    x.doIt();
    return x;
}
```

השימוש בפונקציה נעשה באופן הבא:

```
X x;
int v1, v2;
//...
x << v1 << f << v2;
```

כאשר המהדר רואה את התו f הוא 'מבין' שלפניו שם פונקציה. כלומר, כתובת הפונקציה נתונה לו. לפיכך, מופעל האופרטור שמקבל מצביע לפונקציה כזו. הואיל והאופרטור מחזיר ייחוס לאובייקט מסוג X, אפשר להפעילו בשרשרת כזו. האופרטור מפעיל את הפונקציה שמועברת לו ומחזיר את התוצאה שלה.

## 4.8.2 מניפולטורים במערכת קלט/פלט

במערכת קלט/פלט יש מספר מניפולטורים (מְטַפְּלְלִים) הפועלים על אובייקטים שונים. למשל, יש מניפולטורים הפועלים על אובייקטי פלט כמו: ofstream או fstream. מניפולטורים אחרים פועלים על אובייקטי פלט, או קלט.

לדוגמה:

- **oct** - ממיר את האובייקט למבנה אוקטלי (על בסיס 8). פועל על פלט וקלט.
- **dec** - ממיר את האובייקט למבנה עשרוני (על בסיס 10). פועל על פלט וקלט.
- **hex** - ממיר את האובייקט למבנה (על בסיס 16). פועל על פלט וקלט.
- **endl** - שולח תו של שורה חדשה לאובייקט וכותב את החוצץ הפנימי שלו. פועל על אובייקטי פלט בלבד.
- **ends** - שולח תו '\0' המסמן את סוף המחרוזת. מניפולטור זה פועל על פלט בזיכרון ostrstream.
- **flush** - כותב את החוצץ הפנימי של האובייקט למערכת הקבצים.

כאשר משתמשים באובייקטי הפלט של הספרייה הסטנדרטית, נרשמים הנתונים לחוצץ פנימי של האובייקט תחילה. רק כאשר החוצץ הפנימי מלא, מבוצעת הכתיבה למערכת הקבצים. כדי לכתוב את החוצץ לפני שהוא מתמלא, אפשר להשתמש בפעולות כמו אלו:

```
ofstream out("file.out");
out << 5 << 3 << flush;

out << 5 <<", " << 4 << endl;
```

בפעולת הפלט הראשונה נכתבים המספרים 3 ו-5. לאחר מכן, נכתב החוצץ הפנימי לקובץ. בפעולת הפלט השנייה נכתבים המספרים 4 ו-5, לאחריהם מוכנס התו המציין שורה חדשה ומייד לאחר מכן נכתב החוצץ הפנימי לקובץ.

אפשר לשנות הבסיס של המספרים בדרך זו:

```
cout << 12 << hex << " " << 12 << oct << " " << 12;
```

עבור שורה כזו נקבל את הפלט הבא:

```
122 7a 172
```

ראינו, שלאחר הפעלת מניפולטור מסוים תבוצע המרה על פי בסיס המספרים של אובייקט הפלט, או הקלט.

## 4.9 סיכום

בפרק זה למדנו על ספריית הקלט/פלט הסטנדרטית של ++C. ספרייה זו אינה חלק מהשפה אבל מסופקת יחד עם המהדר. לספרייה הסטנדרטית הוגדר ממשק ANSI, לכן כדאי להשתמש בה ולמנוע קשיי התמרה של תוכניות מפלטפורמה לפלטפורמה.

ספרייה זו בטוחה יותר מהספרייה הדומה לה ב-C. כאן אין פונקציות בעלות מספר וסוג לא ידוע של פרמטרים. המהדר יכול לגלות שגיאות מסוג זה כבר בזמן ההידור ולא רק בזמן הריצה.

לספריית הקלט/פלט (או זרמי קלט/פלט, iostream) יש אובייקטים הכותבים לקבצים, ויש גם אובייקטים המאפשרים לקרוא מקבצים, ויש אובייקטים שמאפשרים קריאה וכתיבה בו-זמנית. בנוסף, יש אובייקטים (ostrstream) המאפשרים להמיר נתונים לתוך חוצצים בזיכרון. אובייקט אחר, istrstream, מאפשר לקרוא מתוך חוצץ בזיכרון. אובייקט מסוג strstream מאפשר להמיר נתונים בעת קריאה, או כתיבה.

ראינו שיש מניפולטורים הפועלים על אובייקטים כאלה ומאפשרים, בצורה נוחה, לשנות מצב של זרם (קלט או פלט). מניפולטורים כאלה מאפשרים למשל לשלוח את הנתונים אשר נאגרו בחוצצים הפנימיים של האובייקטים לקבצים.

תודות ליכולת של ++C להעמסה של פונקציות והחזרת ייחוס (reference) של אובייקטים מפונקציה, ראינו שאפשר לבצע מספר פעולות פלט או קלט במשפט אחד של השפה.

## 4.10 שאלות

1. ממש פונקציה המעתיקה קובץ מקור לקובץ יעד תוך שימוש בפונקציות read ו-gcount.
2. כתוב תוכנית המדפיסה את מספר המילים, התווים והשורות שנמצאות בקובץ מסוים.
3. כתוב תוכנית המוצאת מילה מסוימת בקובץ נתון ומדפיסה את מספר המופעים של המילה בקובץ זה.
4. כתוב תוכנית הקוראת קובץ מסוים ומצמצמת את כל הרווחים בו.
5. כתוב תוכנית אשר מקבלת שתי מילים ושם קובץ. היא תסרוק את הקובץ כדי לגלות בו את המופעים של המילה הראשונה ותחליף אותה במילה השנייה. למשל: change w1 w2 file תשנה את כל המילים "w1" למילה "w2" בקובץ file.

## פרק 5

# ירושה ככלי לשימוש חוזר בקוד

בפרק זה נלמד על מושג **הירושה** ב-++C (inheritance). נראה שירושה היא כלי בעל עוצמה רבה לשימוש והרחבה של תוכנה קיימת, נכיר סוגים שונים של ירושה ב-++C ונלמד את המשמעות של כל סוג. ירושה היא אחד מהתכונות העיקריות של תכנות מוכוון אובייקטים (OOP).

**ירושה** לכשעצמה, היא כלי יעיל אבל לא מספק. השימוש בירושה בא, במקרים רבים, בצירוף לשימוש במנגנון **הפולימורפיזם** (ראה פרק 6). זו הסיבה לחשיבותו של פרק זה בהבנת משמעות **תכנות מוכוון אובייקטים**.

## 5.1 מהי ירושה

במונח **ירושה** (inheritance) אנו נתקלים בחיי היום יום. בדרך כלל, מתייחס המונח לרכוש העובר בירושה מאנשים שהלכו לעולמם. בשפות תכנות מוכוונות אובייקטים מקבל המונח ירושה משמעות שונה לחלוטין. פעמים רבות אנו מתייחסים למושג **ירושה** בהקשר של **הורשת תכונות** מאב או מאם, לבן או לבת. בשפות תכנות מוכוונות אובייקטים המונחים ירושה או הורשה מתייחסים ל**העברת תכונות של עצם (אובייקט) אחד לעצמים אחרים**.

ירושה הוא יחס בין שתי מחלקות, או יותר. כאשר שתי מחלקות משתתפות ביחס ירושה, אזי אומרים שמחלקה אחת **מורישה** ואחת - **יורשת**. ב-++C המחלקה המורישה היא **מחלקה בסיסית** (base class) והמחלקה היורשת היא **מחלקה נגזרת** (derived class). לעיתים מכנים את המחלקה הבסיסית בשם **מחלקת-על** (superclass) ואת המחלקה היורשת - **תת-מחלקה** (subclass).

המחלקה הבסיסית מופשטת ובעלת תכונות כלליות. המחלקה הנגזרת היא מחלקה מפורטת, הכוללת את כל התכונות והפונקציות של המחלקה הבסיסית עם תוספת אפשרית. במחלקה הנגזרת אפשר להוסיף שדות או פונקציות לאלו של המחלקה הבסיסית. לפיכך, ירושה היא מעין הרחבת תכונות אובייקטים.

בניגוד לירושה בין בני אדם, המחלקה הבסיסית "חיה" וקיימת גם אם יש מחלקות אחרות שיורשות ממנה. בדרך כלל, ניתן להגדיר אובייקטים גם מהמחלקה הבסיסית (המורישה).

הירושה, אם כן, היא צורה לשימוש והרחבה פונקציונלית של מחלקה נתונה. המחלקה הנגזרת יכולה להוסיף פרטים רבים למחלקה הבסיסית, מבלי לשנות אותה. תכונה זו חשובה מאוד, כדי לאפשר שימוש חוזר במחלקות נתונות, ובכך לחסוך שִׁכתוב מחדש של קוד.

איך מסמנים ירושה ב-C++? עבור שתי מחלקות, בסיסית ונגזרת (או יורשת), נסמן ירושה באופן הבא (שים לב, Base ו-Derived הם שמות מחלקות, שיכולים להיות שמות כלשהם):

```
class Base { ... };
class Derived : public Base {
...
};
```

**המחלקה הנגזרת** (drived class), Derived, מצוינת על ידי הכנסת הסימן "**:**" לאחר שמה בצירוף מילת המפתח public, ולאחריה שם המחלקה הבסיסית. מילת המפתח יכולה להיות שונה, ובהמשך נראה את המשמעות של האפשרויות השונות.

שורות הקוד שנכתבו למעלה ציינו שהמחלקה Derived יורשת את כל התכונות והפונקציות של המחלקה Base. לכל אובייקט של המחלקה Derived יש את השדות של אובייקט השייך למחלקה Base. כך אפשר להפעיל כל פונקציה של המחלקה Base על אובייקטים של המחלקה Derived.

הדוגמה שראינו זה עתה היא דוגמה ל**ירושה יחידה** (single inheritance). כלומר, המחלקה היורשת יורשת ממחלקה בסיסית אחת ויחידה. לעומת זאת, יש מקרים שירושה יחידה אינה מספקת ויש צורך **בירושה מרובה** (multiple inheritance). ירושה מרובה מציינת שהמחלקה יורשת ממספר מחלקות.

כמו במקרה של ירושה יחידה כך גם בירושה מרובה, מכילה המחלקה היורשת את כל השדות של המחלקות הבסיסיות, עם תוספת אפשרית. בנוסף, כל הפונקציות של המחלקות הבסיסיות, הן פונקציות של המחלקה היורשת.

## 5.2 ירושה יחידה

בירושה יחידה יורשת מחלקה אחת ממחלקה יחידה אחרת. נניח, לדוגמה, שיש לנו מחלקה "אדם" (Person). לכל אדם יש שם, שם משפחה ומספר תעודת זהות. זהו המקרה הבסיסי ביותר של אדם. אם התוכנה שלנו עוסקת במוסד לימודי, כגון אוניברסיטה, נמצא סוגי אנשים שונים, למשל מרצים וסטודנטים. הסטודנטים חייבים בתשלום שכר לימוד שנתי, והמרצים מקבלים משכורת חודשית. לכל אחת מיישויות אלו יש תכונה נוספת, השונה במשמעותה ואינה קיימת במחלקה הבסיסית.

נגדיר את שלוש המחלקות הבאות שמייצגות את היישויות שתוארו כאן.

```
class Person {
    char name[32];
    char familyName[32];
    int  idNumber;

public:
    Person(const char *nm, const char *fnm, int id);
    void print();
};

Person::Person(const char *nm, const char *fnm, int id)
{
    strcpy(name, nm);
    strcpy(familyName, fnm);
    idNumber = id;
}

void Person::print()
{
    cout << name << " " << familyName << " " << id << " ";
}
```

**המחלקה הבסיסית** (base class) Person מספקת את המסגרת הבסיסית לאובייקט המייצג אדם. בבנאי של מחלקה זו מועתקים השם, שם המשפחה ומספר תעודת הזהות.

הפונקציה Print מדפיסה את השדות של אובייקט מסוג Person לפלט הסטנדרטי, כשבין כל שדה יש רווח.

באובייקט מסוג "סטודנט" נוסיף את השדה המציין את שכר הלימוד השנתי שהוא משלם.

```
class Student : public Person {
    int payment;
public:
    Student(const char *nm,const char *fnm,int id,int pay)
       : Person(nm, fnm, id)
    { payment = pay; }
    void print();
};
```

הבנאי של Student מקבל את תכונותיו של אדם בסיסי, ובנוסף להן תכונות וערכים המיוחדים לסטודנטים, כגון שכר הלימוד. הבנאי של **המחלקה הנגזרת** (derived class) מעביר ארגומנטים השייכים לאובייקט הבסיסי לאחר ה-":", ובעזרת קריאה לבנאי המתאים של המחלקה הבסיסית. העברה זו מתבצעת **לאחר** התו המפריד ":".

הקריאה לבנאי הבסיסי המתאים נעשה על ידי שם המחלקה הבסיסית והארגומנטים שמועברים לבנאי המתאים בסוגריים עגולים. מנגנון **העמסה** (overloading) יודע גם כאן למצוא את הבנאי המתאים לפי מספר הארגומנטים וסוגם. במקרה זה, מועברים לבנאי הבסיסי (Person) השם, שם המשפחה ומספר הזהות.

```
void Student::print()
{
    Person::print();
    cout << payment;
}
```

הפונקציה print מדפיסה את השדות של אובייקט מסוג "סטודנט". לאובייקט כזה יש שלושה שדות של האובייקט הבסיסי, ועוד שדה נוסף. כדי להדפיס את שלושת השדות הבסיסים נקראת הפונקציה הבסיסית print, ולאחר מכן נדפיס את השדה הנוסף.

כדי להפעיל את הפונקציה הבסיסית ממחלקה יורשת, עלינו לקרוא לה. לפני שם הפונקציה יש לכתוב את שם המחלקה הבסיסית בתוספת הסימן "::". סימן זה הוא אופרטור המאפשר לקבוע לאיזו מחלקה שייכת הפונקציה הנקראת. הפעלה רגילה של פונקציה בתוך פונקציות של מחלקה יורשת תתייחס **תמיד** לפונקציה של המחלקה היורשת. זו גם הסיבה לכך שאם לא היינו משתמשים באופרטור זה, היתה נקראת הפונקציה print של Student פעם נוספת והיינו מקבלים לולאה אינסופית. רק אם שם הפונקציה הנקראת אינו מתאים לפונקציה במחלקה היורשת, המהדר מנסה למצוא פונקציה כזו במחלקה הבסיסית. בדרך דומה נגדיר את המחלקה "פרופסור":

```
class Professor : public Person {
    int salary;
public:
    Professor(const char *nm,const char *fnm,int id,int sal)
       : Person(nm, fnm, id)
    { salary = sal; }
    void print() {
       Person::print();
       cout << salary;
    }
};
```

ראינו, שאפשר להפעיל פונקציה של מחלקה בסיסית מתוך מחלקה יורשת. הפעלה כזו מאפשרת שימוש בפונקציונליות של המחלקה הבסיסית במחלקה היורשת. מצב זה נקרא **overidding** (דריסה). במקרים רבים אף לא קיים הצורך לכתוב פונקציה במחלקה היורשת. אם, לדוגמה, היתה לנו פונקציה שבודקת אם שני אנשים הם זהים, יכולים היינו לכתוב אותה כך:

```
class Person {
...
      int equal(const Person &p) const
      { return (idNumber == p.idNumber); }
```

```
};

int operator==(const Person &p1, const Person &p2)
{
      return p1.equal(p2);
}
```

הפונקציה משווה בין שני אובייקטים מסוג Person. שני אובייקטים כאלה זהים, אם מספר הזהות שלהם זהה. הוספנו אופרטור גלובלי, שאינו חלק של המחלקה, כך שאפשר יהיה לבדוק את השיוויון בין שני אובייקטים מאותו סוג (במקרה זה - Person) בצורה נוחה.

במקרה של פונקציה זו, אין צורך להוסיף את הפונקציות במחלקות היורשות. מכיון שאובייקטים היורשים מהמחלקה הבסיסית הם אובייקטים בסיסיים, מאפשר המהדר להמיר אובייקט יורש לאובייקט בסיסי. לכן, למשל, מותרת ההמרה של "פרופסור" ל"אדם" ושל "סטודנט" ל"אדם". ליתר דיוק, ההמרה נעשית באופן אוטומטי על ידי המהדר.

לאור האמור, זהו אופן ההפעלה של פונקציות אלו:

```
Student s("leora", "cohen", "3302245", 500);
Professor p("dan", "amir", "220358", 5000);

if (s == p) {
      cout << "something very strange" << endl;
      s.print();
      p.print();
}
```

## 5.2.1 סוג הירושה

עד כה הכרנו **ירושה ציבורית** בלבד, שבה כל השדות והפונקציות בחלק הציבורי של המחלקה הבסיסית נחשבות כציבוריות במחלקה היורשת. סוג הירושה נקבע על ידי מילת המפתח המופיעה לפני שם המחלקה. מילות המפתח הן אחת משלוש המילים:

- **public** - ירושה ציבורית.
- **protected** - ירושה מוגנת.
- **private** - ירושה פרטית.

מילות מפתח אלו גם מתארות אזורי גישה שונים במחלקה. אנו כבר מכירים אזורי גישה **פרטי וציבורי**. אזור הגישה החדש הוא האזור ה**מוגן** (**protected**). אל השדות והפונקציות באזור זה יכולות לגשת מחלקות נגזרות בלבד. לאזור הגישה המוגן מותרת הגישה גם לחברים של המחלקה. למשל:

```
class Base {
protected:
```

```
	int val;
	int f();
...
};
```

לשדה val יכולות לגשת רק מחלקות יורשות (או חברים של המחלקה), ופונקציות או מחלקות אחרות אינן יכולות לעשות זאת. דבר זה תקף גם לגבי פונקציות שנמצאות באזור זה. כלומר, לפונקציה **f** יכולות לגשת מחלקות יורשות, או חברים של המחלקה.

### 5.2.1.1 ירושה פרטית

**בירושה פרטית** (private inheritance) השדות והפונקציות הם פרטיים במחלקה היורשת. אין גישה לשדות ולפונקציות אלו פרט לפונקציות של המחלקה היורשת, או חברים של מחלקה זו. כלומר:

```
class Base {
public:
	int val;
	int f();
	...
};

class Derived : private Base {
...
	void g() { f(); ... }
};

Base b;
Derive d;
d.val = 1;		// error - no access
b.val = 1;		// ok
```

ירושה פרטית שקולה ל**הכלה** (containment), כלומר, אפשר היה להגדיר שדה במחלקה היורשת, שהוא אובייקט מסוג Base ולכן, אובייקט מסוג Drive היה מכיל אובייקט מסוג Base. למרות זאת, קיים הבדל בין הכלה לירושה פרטית.

בפונקציה g של המחלקה היורשת קוראים לפונקציה f. במקרה זה, אין צורך להוסיף קידומת משום ש-f היא פונקציה השייכת למחלקה היורשת. לעומת זאת, אם היינו מגדירים שדה מסוג המחלקה הבסיסית במחלקה היורשת, היה עלינו להשתמש באובייקט המתאים בעת קריאה לפונקציה שלו:

```
class Derived {
	Base b;
...
	void g() { b.f(); ...}
};
```

כמו שרואים בקטע הקוד הזה, הפעלת הפונקציה f נעשית דרך האובייקט. זו הדרך לסרבל את הקוד (מבחינה ויזואלית), לכן, יש מתכנתים המעדיפים להשתמש בירושה פרטית, ולא בהכלה. בהמשך נראה דוגמאות נוספות לירושה פרטית.

## 5.2.1.2 ירושה מוגנת

**ירושה מוגנת** (protected inheritance) מוגדרת בעזרת מילת המפתח **protected**. למילת מפתח זו יש שתי משמעויות. כאשר מילת המפתח מופיעה באזור של המחלקה - האזור הזה, עד לאזור הבא, פתוח לגישה של מחלקות יורשות. למשל:

```
class Base {
private:
      int x1;
protected:
      int x2;
...
};

class Derived : public Base {
...
      void set_x2(int v) { x2 = v; }
};

Derived d;
d.x2 = 1;          // error - no access
d.set_x2(2);       // ok through Derived function
```

לפי קוד זה, נראה שיש אפשרות לשנות ערך לשדות בחלק המוגן, או לגשת לחלק המוגן למחלקות יורשות בלבד. תכונה דומה לכך היא ירושה מוגנת:

```
class Base {
protected:
      int x;
...
};
class D1 : protected Base {
...
      void f(Base *bp, D1 *dp);
};

void D1::f(Base *bp, D1 *dp)
{
      int x1, x2;
      x1 = bp->x;        // error - no access
      x2 = dp->x;        // ok - object is of type D1
}
```

הפונקציה f של המחלקה D1 יכולה לגשת לאובייקטים מסוג D1 בלבד (כולל האובייקט הנוכחי this), אל שדה x שנמצא בחלק המוגן (protected). לפונקציה זו אין גישה לשדות בחלק המוגן של אובייקטים מסוג Base, שאינם מסוג D1.

```
class D2 : public D1 {
...
      void f1(D1 *d, D2 *d2);
}

void D2::f1(D1 *d, D2 *d2)
{
      int x1, x2;
      x1 = d->x;          // error - no access
      x2 = d2->x;         // ok - object is of type D2
}
```

בפונקציה f1 השייכת למחלקה D2 יכולים לגשת לשדה x רק אובייקטים מסוג D2. בפונקציה זו השייכת למחלקה D2, אובייקטים מסוג D1 אינם יכולים לגשת לשדה x. הדרך הקלה לזכור נקודה זו (שגורמת לבלבול אפילו בקרב מתכנתים מנוסים) היא, שאובייקט מסוג D1 **אינו** אובייקט מסוג D2. לכן, **הגישה** (access) המוענקת לחלק השמור של אובייקטים מסוג D2 היא רק לאובייקטים מסוג D2, או ליורשים ממנו. למעשה, אובייקט מסוג D1 יכול להיות מסוג אחר היורש מ-D1, לכן גישה כזו אינה אפשרית.

### 5.2.1.3 הורשה ציבורית

בהורשה ציבורית הופכים השדות הציבוריים של המחלקה הבסיסית לשדות ציבוריים במחלקה היורשת. השדות המוגנים במחלקה הבסיסית הופכים לשדות מוגנים במחלקה היורשת, והשדות הפרטיים של המחלקה הבסיסית הם מחוץ לתחום במחלקה היורשת. ירושה ציבורית מסומנת על ידי מילת המפתח **public**.

```
class D3 : public Base {
...
};

void f(Base b);
D3 d;
f(d);
```

יתרה מזאת, פונקציות הפועלות על אובייקטים מסוג Base פועלות באופן רגיל על אובייקטים מסוג D3. המהדר הוא זה שממיר את האובייקט מסוג D3 לאובייקט מסוג Base. במקרה שהאובייקט מועבר בקריאה רגילה (לא כייחוס) כמו לפונקציה f, מתבצע **חיתוך** (slicing) של האובייקט לאובייקט הבסיסי. כלומר, המהדר מפעיל את בנאי ההעתקה כדי ליצור אובייקט בסיסי מהאובייקט הנתון. אם האובייקט מועבר כייחוס, אין צורך לבצע חיתוך מאחר וייחוס הוא שם אחר לאובייקט.

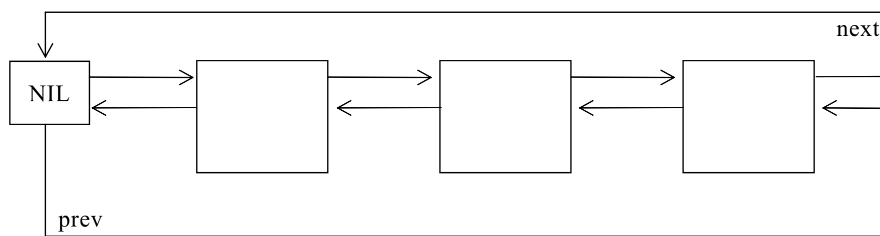
## 5.2.2 דוגמה - רשימה מעגלית כפולה

בסעיף זה נראה דוגמה לרשימה מעגלית כפולה, ונלמד את אופן השימוש בה במחלקות היורשות מהרשימה. הקוד לרשימה המעגלית הכפולה נמצא בקבצים **dlist.h** ו-**dlist.cpp**.

הרשימה מורכבת מצמתים, לכל צומת יש מצביע לצומת הקודם, ומצביע לצומת הבא. לרשימה יש צומת מיוחד המסמן את תחילת הרשימה ונקרא NIL. צומת זה מצביע לראש הרשימה וגם לסופה, לכן הרשימה היא מעגלית.

כדי לממש את הרשימה נגדיר שלוש מחלקות:

- **link** - צומת ברשימה.
- **list_iter** - איטרטור לרשימה המאפשר לסרוק את כל הצמתים ברשימה.
- **list** - אוסף הצמתים המייצג את הרשימה.




***איור 5.1:*** *מבנה הרשימה המעגלית הכפולה*

### 5.2.2.1 צומת ברשימה

צומת ברשימה מכיל שני מצביעים (next ו-prev) שמצביעים לצומת הבא והקודם, בהתאמה. הצומת ברשימה מעניק לרשימה גישה לחלק הפרטי שלו, כדי לאפשר הכנסה והוצאה של צמתים מהרשימה.

```
class link {
friend class list;
      link *prev;
      link *next;
public:
      link();
      ~link();
      link *get_next() { return next; }
      link *get_prev() { return prev; }
};
```

הבנאי של הצומת מאתחל את המצביעים אל האובייקט הנוכחי, this.

```
link::link()
{
      next = prev = this;
}

link::~link()
{
}
```

המפרק של הצומת ריק כעת. עובדה בולטת היא שלצומת אין עדיין כל נתונים, פרט למצביעים, ומייד נראה כיצד נוכל בעזרת ירושה להכניס בו נתונים. הפונקציות get_next ו-get_prev מאפשרות גישה לצומת הבא או הקודם, בהתאמה, ונועדו לשימוש האיטרטור. מכיון שלפנינו רשימה כפולה, יכולה האיטרציה להיות דו- כיוונית, כפי שנראה בהמשך.

## 5.2.2.2 האיטרטור

**האיטרטור** (iterator) מאפשר לעבור על צמתים ברשימה, או פשוט - **לסרוק אותם**, ולכן אפשר לכנותו **סורק**. כדי לעבור על צמתים יש לאיטרטור שדה שמציג את הצומת הנוכחי, והוא מקבל אליו מצביע. האיטרטור מאפשר לעבור על הרשימה בשני הכיוונים, מראש הרשימה לסופה ולהיפך. מעבר כזה על הרשימה יכול לעבור דרך הצומת, שהוא שורש הרשימה. לכן, כשעוברים על הרשימה יש לשים לב לכך במיוחד.

כדי לקבל את הצומת הנוכחי, מגדיר האיטרטור אופרטור מסוג "*". בעזרתו נראה האיטרטור כמצביע, שכן כשמשתמשים באופרטור "*" מקבלים את תוכן המצביע, שבמקרה זה הוא הצומת הנוכחי שאליו מצביע האיטרטור.

```
class list_iter {
      link *current;
public:
      list_iter(link *l) { current = l; }
      link *operator*()
      { return (current); }
      link *next();
      link *prev();
      list_iter &operator++() { next(); return *this; }
      list_iter operator++(int);
      list_iter &operator--() { prev(); return *this; }
      list_iter operator--(int);
      int operator==(const list_iter &i) const
      { return (current == i.current); }
      int operator!=(const list_iter &i) const
      { return (current != i.current); }
};
```

הפונקציה next מקדמת את המצביע של הצומת הנוכחי למצביע הבא, בעזרת קריאה לפונקציה get_next של הצומת הנוכחי. הפונקציה prev מחזירה את המצביע הנוכחי צומת אחד לאחור.

```
link *list_iter::next()
{
      current = current->get_next();
      return current;
}

link *list_iter::prev()
{
      current = current->get_prev();
      return current;
}
```

ראינו שיש שני אופרטורים מסוג "++" המוגדרים עבור האיטרטור (iterator). אופרטורים אלה מקדמים את האיטרטור לצומת הבא ברשימה. האופרטור שאינו מקבל פרמטר מקדם את האיטרטור תחילה, ולאחר מכן מחזיר ייחוס לאיטרטור (**prefix operator**). הפעלת האופרטור האחרון היא במשפטים מסוג זה:

```
list_iter i(lnk);
++i;
```

האופרטור השני, שמקבל שלם, מקדם את האיטרטור, אבל מחזיר איטרטור שמתאים למצב הקודם, לפני הקידום. אופרטור זה נקרא **postfix operator** והוא דומה לאופרטור המגדיל שלם, אבל משתמש בערכו הקודם:

```
i++;
```

הפרמטר שמקבל האופרטור נועד לציין למהדר שזה אופרטור **postfix** ויש להפעילו במשפטים המתאימים. כדי לממש את הסמנטיקה המבוקשת, מעתיק האופרטור את האובייקט הנוכחי לאובייקט זמני, מקדם את האיטרטור הנוכחי, ומחזיר את האובייקט הזמני.

```
list_iter list_iter::operator++(int)
{
    list_iter tmp(current);
    next();
    return tmp;
}
```

באופן דומה, מוגדר האופרטור "--" (אופרטור החיסור אשר מממש תזוזה לאחור). ראוי לזכור שיש שני אופרטורים כאלה, postfix ו-prefix, בהתאמה.

```
list_iter list_iter::operator--(int)
{
      list_iter tmp(current);
      prev();
      return tmp;
}
```

האופרטורים שמחזירים איטרטור המתאים למצב הקודם (postfix) פחות יעילים, משום שאובייקט המוחזר מהאופרטור מועבר לא כייחוס, אלא אל המחסנית של התוכנית. בנוסף, מגדיר האופרטור אובייקט זמני שאליו הוא מעתיק את האובייקט הנוכחי לפני ההעתקה. אופרטורים אלה מבצעים שתי העתקות, הראשונה לאובייקט הזמני בתוך האופרטור, והשנייה לאובייקט המוחזר אל המחסנית. מומלץ, אם כן, להשתמש באופרטורים האחרים במידת האפשר.

### 5.2.2.3 הרשימה

הרשימה מכילה צומת אחד, המציין את ראש הרשימה והוא ריק (אינו מכיל נתונים). צומת זה מציין גם את סוף הרשימה. בנוסף, יש לרשימה שדה שמציין את מספר הצמתים ברשימה.

```
class list {
      link *NIL;         // the NIL   node
      int nobjects;      // number of objects
      void put_after_self(link *self, link *lnk);
      void put_before_self(link *self, link *lnk);
      void unlink(link*);
public:
      list();
      ~list();
      void append(link *);
      void insert(link *);
      void insert(link *pos, link *item)
      { put_before_self(pos, item); }
      link *head();            // gets the head
      link *tail();            // gets the tail
      link *remove_head();
      link *remove_tail();
      void remove(link *);
      const link *nil() const { return NIL; }
      list_iter begin() { return list_iter(head()); }
      list_iter end()   { return list_iter(NIL); }
};
```

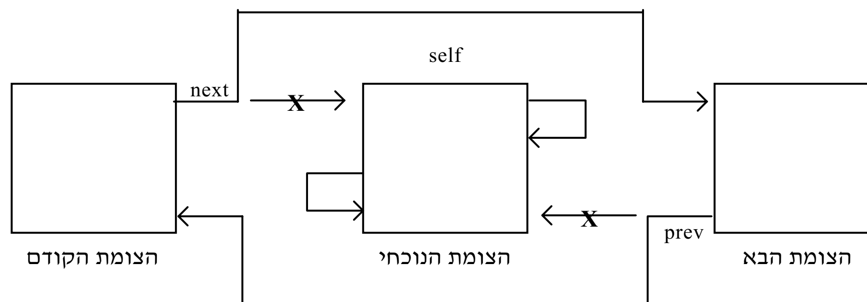
הפונקציה put_after_self נמצאת בחלק הפרטי של המחלקה ומקבלת מצביעים לשני צמתים. הצומת הראשון נמצא ברשימה, הצומת השני הוא צומת שמכניסה הפונקציה לרשימה. הפונקציה מכניסה את הצומת השני לאחר הראשון.

```
void list::put_after_self(link *self, link *lnk)
{
      lnk->prev = self;
      lnk->next = self->next;
      self->next->prev = lnk;
      self->next = lnk;
}
```

הפונקציה put_before_self מקבלת אף היא מצביעים לשני צמתים (כמו הפונקציה הקודמת). הראשון מביניהם הוא צומת שנמצא ברשימה, והשני הוא הצומת שיש להוסיף לרשימה. פונקציה זו מכניסה את הצומת השני לפני הראשון.

```
void list::put_before_self(link *self, link *lnk)
{
      lnk->next = self;
      lnk->prev = self->prev;
      self->prev->next = lnk;
      self->prev = lnk;
}
```

הפונקציה unlink מנתקת צומת משני הצמתים השכנים לו. הפונקציה מעדכנת את המצביע prev של הצומת הבא שיצביע לצומת הקודם, (במקום אל הצומת הנוכחי). בנוסף, היא מעדכנת את המצביע next של הצומת הקודם לצומת הנוכחי, כדי שיצביע אל הצומת הבא (שאחרי הצומת הנוכחי). לאחר פעולת הפונקציה, המצביעים של הצומת המנותק (זה הצומת הנוכחי), מצביעים אל הצומת עצמו.




```
void list::unlink(link *self)
{
      self->next->prev = self->prev;
      self->prev->next = self->next;
      self->next = self->prev = self;
}
```

**בנאי הרשימה** מקצה את הצומת הריק (NIL) הראשון של הרשימה, והמפרק משחרר את הצומת הריק. הבנאי גם מאפס את מספר הצמתים ברשימה. גישה אחרת היא להגדיר צומת אחד בתוך הרשימה, ולחסוך את ההקצאה והשחרור של הצומת הריק.

```
list::list()
{
      NIL = new link;
      nobjects = 0;
}
list::~list()
{
```

```
	delete NIL;
}
```

הפונקציה append מגדילה את מספר הצמתים ברשימה ומכניסה צומת בסופה. כדי להכניס צומת בסוף הרשימה הוא מוכנס **לפני** הצומת הריק. מכיון שזו רשימה מעגלית, שקול הדבר להכנסת הצומת בסוף הרשימה.

```
void list::append(link *lnk)
{
	nobjects++;
	put_before_self(NIL, lnk);
}
```

הפונקציה insert מכניסה צומת בתחילת הרשימה ומגדילה את מספר הצמתים ברשימה. הכנסת צומת לאחר הצומת הריק שקולה להכנסה בתחילת הרשימה.

```
void list::insert(link *lnk)
{
	nobjects++;
	put_after_self(NIL, lnk);
}
```

שתי הפונקציות, head ו-tail, מחזירות מצביעים לראש ולזנב הרשימה, בהתאמה.

```
link *list::head()
{
	return NIL->next;
}

link *list::tail()
{
	return NIL->prev;
}
```

הפונקציה remove_head מוחקת את **הצומת הראשון** מהרשימה. לאחר המחיקה מקטינה הפונקציה את מספר הצמתים שברשימה.

```
link *list::remove_head()
{
	link *lnk = 0;
	 if (nobjects > 0) {
	      lnk = NIL->next;
	      unlink(lnk);
	      nobjects--;
	 }
	 return lnk;
}
```

הפונקציה remove_tail מוחקת את **הצומת האחרון** ברשימה. לאחר המחיקה מקטינה הפונקציה את מספר הצמתים שברשימה.

```
link *list::remove_tail()
{
	link *lnk = 0;
	if (nobjects > 0) {
		lnk = NIL->prev;
		unlink(lnk);
		nobjects--;
	}
	return lnk;
}
```

הפונקציה remove מוחקת **צומת נתון** מהרשימה, ולכן קטן מספר הצמתים ברשימה.

```
void list::remove(link *lnk)
{
	if (nobjects > 0) {
		unlink(lnk);
		nobjects--;
	}
}
```

## 5.2.2.4 שימוש ברשימה בעזרת ירושה

אין שימוש רב לרשימה המורכבת רק מ-link. אם רוצים להכניס נתונים לרשימה, ניתן לבצע זאת בעזרת ירושה. למשל, אנו יכולים להגדיר צומת המכיל שלם, ויורש מהצומת הבסיסי של הרשימה:

```
class int_link : public link {
	int val;
public:
	int_link(int v) { val = v; }
	operator int() { return val; }
};
```

מכיון שהצומת int_link יורש בירושה ציבורית מהצומת הבסיסי של הרשימה, אפשר להשתמש במצביע לאובייקט מסוג זה בכל מקום בו, אנו משתמשים במצביע לאובייקט הבסיסי. אובייקט מסוג int_link הוא בעל אותם השדות והפונקציות של link, בתוספת לשדה השלם שלו ו**אופרטור ההמרה** (conversion operator) לשלם.

### 5.2.2.4.1 הכנסת אובייקטים לרשימה

אפשר להעביר לרשימה מצביעים לאובייקטים מסוג int_link, ולכן אנו יכולים להכניס אובייקטים כאלה בתחילת הרשימה ובסופה:

```
void append(list &lst, int l, int h)
{
	for (int i = l; i <= h; i++) {
```

```
		lst.append(new int_link(i));
	}
}

void insert(list &lst, int l, int h)
{
	for (int i = l; i <= h; i++) {
		lst.insert(new int_link(i));
	}
}
```

הפונקציה append מכניסה אובייקטים **בסוף** הרשימה, והפונקציה insert מכניסה אובייקטים **בראש** הרשימה. בשני המקרים מקבלת הפונקציה תחום להכנסת האובייקטים, וכל האובייקטים בתחום מוכנסים לרשימה בקצה המתאים.

### 5.2.2.4.2 הוצאת אובייקטים מהרשימה

כדי להוציא אובייקטים מהרשימה, בתחילתה או בסופה, יש להשתמש בפונקציות המתאימות. שתי הפונקציות, remove_head ו-remove_tail, מרחיקות אובייקטים מהרשימה בקצה המתאים. מספר האובייקטים המורחק הוא n, וזהו פרמטר של הפונקציה.

```
void remove_head(list &lst, int n)
{
	for (int i=0; i< n; i++)
		lst.remove_head();
}

void remove_tail(list &lst, int n)
{
	for (int i=0; i<n; i++)
		lst.remove_tail();
}
```

### 5.2.2.4.3 מעבר על הרשימה

עד כה הכנסנו או מחקנו אובייקטים מהרשימה ללא צורך בהמרת מצביעים. בעת מעבר על רשימה יש להגדיר איטרטור שמחזיר מצביעים לאובייקטים מסוג link, למרות שהכנסנו לרשימה אובייקטים מסוג int_link. במקרה זה, עלינו להמיר את המצביע למצביע נכון.

הפונקציה הבאה משתמשת באיטרטור כדי לעבור על הרשימה. בלולאה הראשונה עוברת הפונקציה על הרשימה מתחילתה לסופה. כדי לקדם את האיטרטור לאובייקט

הבא ברשימה משתמשת הפונקציה באופרטור לאחר השימוש באופרטור זה מופעל אופרטור המצביע "*" (derefernce operator), כדי לקבל את האובייקט. במקרה זה, יש להמיר את סוג המצביע לסוג המתאים (int_link). המרה כזו היא אחד החסרונות של השיטה. בהמשך נראה כיצד ניתן להתגבר על חיסרון זה.

התנאי של האיטרציה הוא שהאיטרטור שונה מהאיטרטור שמסמן את סוף הרשימה. תנאי זה נבדק בחלק התנאי של הלולאה.

בשלב השני משתמשת הפונקציה באופרטור "--" המקדים (prefix) של האיטרטור. בשלב זה עוברת הפונקציה על הרשימה בכיוון ההפוך, מסופה לתחילתה.

```
void print(list &lst)
{
	list_iter i(lst.begin());
	int_link *cur;
	list_iter ei(lst.end());
	cout << "\n(";

	while (i != ei) {
		cur = (int_link*)*(i++);
		cout << *cur << " ";
	}
	cout << ")       " << endl << "(";
	cur = (int_link*)*(--i);
	while (i != ei) {
		cout << *cur << " ";
		cur = (int_link*)*(--i);
	}
	cout << ")       "  << endl;
}
```

התוכנית הראשית בקובץ **list_t.cpp** מנצלת את כל הפונקציות שתוארו עד כה, כדי להכניס אובייקטים לרשימה ולהדפיס את האובייקטים הנמצאים בה. הפונקציה גם מבטלת חמישה אובייקטים מתחילת הרשימה ומסופה.

```
int main()
{
	list li;
	append(li, 0, 10);
	insert(li, 10, 20);
	print(li);
	remove_head(li, 5);
	remove_tail(li, 5);
	print(li);
	return 0;
}
```

הפלט של תוכנית זו ייראה כך:

```
(10 11 12 13 14 15 16 17 18 19 20 0 1 2 3 4 5 6 7 8 9 10)
(10 9 8 7 6 5 4 3 2 1 0 20 19 18 17 16 15 14 13 12 11 10)

(15 16 17 18 19 20 0 1 2 3 4 5)
(5 4 3 2 1 0 20 19 18 17 16 15)
```

## 5.2.3 ירושה מהרשימה

כשנתונה המחלקה list אנו יכולים ליצור מחלקות חדשות שתשתמשנה בפונקציונליות של מחלקה זו. אנחנו יכולים, לדוגמה, ליצור **מחסנית** (stack) **ותור** (queue).

### 5.2.3.1 מחסנית

**מחסנית** (stack) היא מבנה נתונים המכיל אובייקטים אחרים. כאשר מכניסים אובייקט למחסנית הוא מוכנס בראש המחסנית, כאשר מוציאים אובייקט מהמחסנית הוא יוצא מראשה. כלומר, האובייקט **האחרון שהוכנס** הוא **הראשון שייצא** (LIFO). אם נרשום זאת כמשוואה, נקבל:

```
pop(push(x)) = x
```

הפעולה **push** "דוחפת" אובייקט למחסנית, והפעולה **pop** מוציאה אובייקט מהמחסנית. כדי לממש את המחלקה מחסנית נגדיר אותה כיורשת ירושה פרטית מהמחלקה מסוג "רשימה" (list). נבצע זאת בדרך הבאה:

```
class stack : private list {
public:
      stack() {}
      void push(link *l) { insert(l); }
      link *pop() { return remove_head(); }
};
```

לצורך מימוש המחסנית דרושות שתי שורות קוד, שמפנות את הקריאות לפונקציות הרשימה! כלומר, בעבודה מועטה ביותר ובעזרת שימוש בירושה פרטית קיבלנו מבנה נתונים חדש. שים לב, השתמשנו **בירושה פרטית**, לכן שאר הפונקציות הציבוריות של המחלקה list הן מחוץ לתחום למשתמש במחלקה זו. מאחר והירושה פרטית, המשתמש במחסנית אינו יכול להשתמש בפונקציות של הרשימה כמו הכנסה של אובייקטים לסופהה. הדבר תואם את הגדרת המחסנית, שבה אין אפשרות להכניס אובייקטים לתחתית המחסנית, או להוציא אובייקטים מתחתית המחסנית.

### 5.2.3.2 תור

**תור** (queue) הוא מבנה נתונים אשר בו האובייקט **הראשון שמוכנס הוא גם הראשון שיוצא** (FIFO). תור במדעי המחשב דומה לתור בסופרמרקט בעמדת התשלום, כאשר הראשון להגיע מקבל את השירות הראשון. גם מחלקה זו ניתנת למימוש בקלות בעזרת הרשימה וירושה פרטית.

```
class queue : private list {
public:
      queue() {}
      void put(link *l) { append(l); }
      link *get()       { return remove_head(); }
};
```

מכיון שהשתמשנו **בירושה פרטית**, אין למתכנת המשתמש במחלקה זו גישה לשאר הפונקציות הציבוריות של הרשימה. אם היתה למשתמש גישה לשאר הפונקציות של הרשימה, הוא היה יכול להפר את חוקיות התור. לכן, במקרים כאלה יש להשתמש בירושה פרטית.

## 5.2.4 ניהול זיכרון והגדרת אופרטורים

בפרק 3 למדנו על מחלקות וראינו שאפשר **להעמיס אופרטורים** (operator overloading). בין שאר האופרטורים שניתן להגדיר למחלקה, הכרנו את האופרטור **new** ואת האופרטור **delete**. אלו הן פונקציות סטטיות לפי ברירת המחדל, שאינן קשורות למחלקה מסוימת. כאשר מפעילים את אופרטור הקצאת הזיכרון new, עדיין אין אובייקט, ולכן הוא אינו קשור לאובייקט מסוים. בדומה לכך, כשמפעילים את אופרטור השחרור, האובייקט כבר פורק, לכן גם אופרטור זה אינו יכול להיות קשור לאובייקט מסוים.

בסעיף זה נראה כיצד להגדיר אופרטורים אלה עבור מחלקות. נניח, שקיימות שתי מחלקות הכתובות כך :

```
class Base {
        int b;
public:
        Base(int bo) { b = bo; }
        void *operator new(size_t sz)
        { return ::operator new(sz); }
        void operator delete(void *p)
        { ::delete(p); }
};

class Derive : public Base {
        int d;
public:
        Derive(int d0, int bo) : Base(bo) { d = d0; }
};
```

למחלקה הבסיסית יש שדה שלם אחד וגם למחלקה היורשת ממנה יש שדה שלם אחד. המחלקה הבסיסית מגדירה את האופרטורים של הקצאת הזיכרון.

### 5.2.4.1 הגדרת אופרטור new

כאשר מחלקה רוצה לטפל בניהול הזיכרון ובהקצאת אובייקטים שלה, על המתכנת להגדיר עבורה את אופרטור הקצאת הזיכרון **new**. דרך כתיבת האופרטור יכולה להיות אך ורק כמו בדוגמה הקודמת. האופרטור יכול לקבל ארגומנט אחד בלבד מסוג size_t (למעשה, זה שלם חסר סימן). ארגומנט זה מציין את גודל הזיכרון הנדרש להקצאה. האופרטור מחזיר מצביע מסוג void שהוא מצביע לאזור הזיכרון שהוקצה.

הארגומנט שמציין את גודל הזיכרון המבוקש נראה מיותר במבט ראשון, שהרי גודל אובייקט של המחלקה ידוע לפונקציה זו, הוא זהה ל-sizeof(*this). אבל, כאשר מגדירים אופרטור כזה, הוא תקף גם עבור מחלקות נגזרות. על כן, כשמקצים אובייקט של מחלקה נגזרת, ייקרא אופרטור זה עם גודל שיכול להיות שונה מגודלו של אובייקט בסיסי. לדוגמה:

```
Base *bp1, *bp2;
bp1 = new Base(1);
bp2 = new Derive(1, 2);
```

בשתי ההקצאות בקטע קוד זה ייקרא האופרטור שהוגדר במחלקה הבסיסית. גודלו של הזיכרון המבוקש יהיה שונה בכל מקרה.

### 5.2.4.2 הגדרת האופרטור delete

תפקיד האופרטור **delete** לשחרר זיכרון שהוקצה קודם לכן על ידי new. כמו במקרה הקודם, אפשר להגדיר אופרטור זה כפי שהצגנו בדוגמה. כלומר, האופרטור מקבל מצביע מסוג void, המצביע לאזור הזיכרון שיש לשחרר. כמו במקרה הקודם, אופרטור זה עובר בירושה. במילים אחרות, כאשר מפעילים את האופרטור delete על מצביעים למחלקות יורשות, ייקרא אופרטור זה אם הוגדר כפונקציה במחלקה הבסיסית, או אם הוגדר במחלקה היורשת. כלומר, הפעלת delete על אובייקטים של מחלקה יורשת תגרום להפעלת האופרטור שהוגדר במחלקה הבסיסית. לפיכך, לא יופעל האופרטור הגלובלי לשחרור זיכרון של המערכת. למשל:

```
delete bp1;
delete bp2;
```

בשני המקרים ייקרא האופרטור delete של המחלקה הבסיסית.

### 5.2.4.3 שימושים באופרטורי ניהול הזיכרון

כאשר מוגדרים האופרטורים לניהול זיכרון עבור מחלקה מסוימת, תיעשה הקצאה ושחרור של אובייקטים בעזרתם. בדוגמה הקודמת ראינו שאופרטורים אלה השתמשו באופרטורים הגלובלים של המערכת. לשימוש מסוג זה אין שום יתרון. מתי אם כן שימושי להעמיס את אופרטורי הניהול של זיכרון? כדי לענות על השאלה, נניח שיש לנו מחלקה שמייצגת נקודה במרחב דו-מימדי.

המחלקה מוגדרת כך:

```
class Point {
      int x,y;
...
};
```

נניח, שאנו מקצים אובייקטים רבים כאלה בתוכנית. לכל הקצאה של אובייקט כזה שומרת המערכת מידע ניהולי נוסף, כמו למשל גודל אזור הזיכרון שהוקצא. במקרה כזה, אם אנו מבקשים הקצאה שמספיקה לשני שלמים, תקצה המערכת זיכרון לשלושה שלמים. באחד מהם תשמור המערכת את גודל אזור הזיכרון המוקצה. בדרך זו יש בזבוז של חמישים אחוז בכמות הזיכרון!

בנוסף לבעיה הקודמת, צריך האלגוריתם להקצאת זיכרון לחפש אזור זיכרון מתאים לבקשת ההקצאה. אזור כזה חייב להיות מספיק גדול כדי לספק את הדרישה. מאחר וכך, אלגוריתם ההקצאה הכללי יבזבז זמן בכל הקצאת זיכרון, כדי למצוא את בלוק הזיכרון המתאים.

אפשר לשפר מצב זה, מפני שאנו יודעים בדיוק את גודל אזור הזיכרון המבוקש. אם נגדיר מאגר זיכרון המיועד למחלקה זו בלבד ובו מספר רב של בלוקי זיכרון, נוכל לצרוך ממנו את מספר בלוקי הזיכרון הדרושים לנו בכל פעם. אם נוצלו כבר כל הבלוקים במאגר, נקצה בלוקים נוספים. מכיון שכל הבלוקים הם באותו גודל, אין צורך לחפש בלוק בגודל מתאים, ולכן נחסוך זמן חיפוש. בנוסף, בזבוז הזיכרון במערכת יהיה קטן יותר, כי מקצים מספר גדול של בלוקים, ומידע הניהול נשמר פעם אחת עבור כולם יחד. כדי להקצות זיכרון מאגר, נלקח הבלוק הראשון הפנוי במאגר, ולכן אין צורך לחפש בלוק מתאים, משום שכולם באותו גודל! בדרך זו ניתן להקטין את כמות הזיכרון הנדרש מהמערכת, ולהגדיל את מהירות העבודה שלה.

```
class Pool {
   char *memptr;
   int elemsize;   // size of elelment
public:
   ...
   void *alloc()
   { void *res = memptr; memptr = memptr->next; return res;}
   ...
};
```

כתרגיל, אני משאיר לך להגדיר מחלקה כמו המחלקה Pool, שמקצה בלוקי זיכרון בגודל קבוע.

## 5.3 ירושה מרובה

בסעיף זה נעסוק בירושה מרובה. **ירושה מרובה** (multiple inheritance) היא כזו בה **מחלקה** יכולה לרשת ממספר מחלקות. בשפות מונחות אובייקטים אחרות אין תמיכה בירושה מרובה ואכן, בתחילת דרכה לא תמכה בה גם שפת C++.

בירושה מרובה נמצא גם כן סוגי ירושה ציבורית, מוגנת ופרטית, כמו בירושה יחידה. המשמעות של סוגי ירושה אלה היא כמו בירושה יחידה.

### 5.3.1 הגדרת ירושה מרובה

כשרוצים להגדיר מחלקה היורשת ממספר מחלקות, יש לפרטן ברשימת הירושה בדומה לירושה יחידה. לכל מחלקה בסיסית יש לציין את סוג הירושה בעזרת אחת ממילות המפתח: private, protected ו-public.

נניח, למשל, שיש לנו שתי מחלקות. הראשונה מייצגת רכב ימי בלבד, כגון ספינות או סירות. המחלקה השנייה מייצגת רכב יבשתי בלבד, כגון מכוניות או אופנועים. כשרוצים לייצג רכב אמפיבי בעזרת מחלקה, אנחנו יודעים שרכב כזה הוא רכב ימי וגם רכב יבשתי. לכן, המחלקה "רכב אמפיבי" צריכה לרשת מהמחלקה "רכב ימי" ומהמחלקה "רכב יבשתי". כדי לציין עובדה זו נפרט את הירושות ברשימת **המחלקות הבסיסיות** (initialization list) של המחלקה "רכב אמפיבי".

```
class WaterVehicle {
    int maxspeed;         // maximum speed
...
public:
...
    WaterVehicle(int ms) { maxspeed = ms; }
    int get_maxspeed() const { return maxspeeed; }
};

class LandVehicle {
    int maxspeed;         // maximum speed
...
public:
...
    LandVehicle(int ms) { maxspeed = ms; }
    int get_maxspeed() const { return maxspeeed; }
};


class AmphibiousVehicle
    : public WaterVehicle,
```

```
    public LandVehicle {
...
};
```

כאשר מגדירים את המחלקה המייצגת רכב אמפיבי כיורשת מהמחלקה המייצגת רכב ימי ומהמחלקה המייצגת רכב יבשתי, אנו מעניקים לכל אובייקט מסוג רכב אמפיבי את התכונות של רכב ימי ורכב יבשתי. כלומר, המחלקה רכב אמפיבי תכיל את כל השדות והפונקציות של רכב ימי, ובנוסף לכך, יהיו בה כל השדות והפונקציות של רכב יבשתי. ומעבר לכל אלה, יש לו את כל התכונות הנוספות המיוחדות לרכב אמפיבי, ואשר אינן שייכות למחלקות הבסיסיות.

לאובייקט מסוג רכב אמפיבי יש עתה שתי פונקציות בשם get_maxspeed() שמחזירות ערך שלם. מצב כזה אינו מוגדר, או ברור. כשרוצים לקרוא לפונקציה get_maxspeed() עבור אובייקט מסוג רכב אמפיבי, איזו מהן תיקרא?

```
AmphibiousVehicle av;
int speed = av. get_maxspeed();  // error - ambiguity
function call
```

בירושה מרובה יכול להיווצר מצב של חוסר ודאות באשר לפונקציות, או שדות בעלי אותם השמות. אפשר להתגבר על כך בצורה הבאה:

```
speed = av::WaterVehicle.get_maxspeed();
```

במשפט זה אנו מקבלים את מקסימום המהירות (maxspeed) על ידי קריאה מפורשת לפונקציה של רכב ימי. הדבר נעשה על ידי **אופרטור הבחירה** "::" (scope resolution operator), שמציין את המחלקה הבסיסית המבוקשת.

דרך אחרת להתגבר על בעיה זו, היא להגדיר פונקציה מתאימה במחלקה היורשת.

```
class AmphibiousVehicle
      : public WaterVehicle, public LandVehicle {
...
public:
...
      int get_maxspeed() const
      { return LandVehicle::get_maxspeed(); }
      int get_maxwaterspeed() const
      { return WaterVehicle::get_maxspeed(); }
};
```

עתה הגדרנו שתי פונקציות, כשהפונקציה המחזירה את המהירות המקסימלית מחזירה את המהירות המקסימלית של רכב יבשתי. הפונקציה השנייה מחזירה את המהירות המקסימלית של רכב ימי.

## 5.3.2 אתחול של מחלקות בסיסיות

כאשר הבנאים של המחלקות הבסיסיות מקבלים פרמטרים, יש להעביר אותם דרך הבנאי של המחלקה היורשת. לדוגמה, אם הבנאים של המחלקה הבסיסיות מקבלים מהירות מקסימלית, כך:

```
class WaterVehicle {
...
      int max_speed;
public:
      WaterVehicle(int ms) { max_speed = ms; }
...
};

class LandVehicle {
...
      int max_speed;
public:
      LandVehicle(int ms) { max_speed = ms; }
...
};

class AmphibiousVehicle
      : public WaterVehicle, public LandVehicle {
...
public:
      AmphibiousVehicle(int mws, int mls);
...
};

AmphibiousVehicle::AmphibiousVehicle(int mws, int mls)
      : WaterVehicle(mws), LandVehicle(mls)
{
      ...
}
```

אנו מעבירים את הפרמטרים למחלקות הבסיסיות ברשימת האתחול של הבנאי במחלקה היורשת. בדוגמה זו מקבל הבנאי של המחלקה היורשת מהירות מקסימלית כאשר הרכב נע בים ומהירות מקסימלית כאשר הרכב נע ביבשה. ייתכנו מקרים בהם המחלקה היורשת אינה מקבלת פרמטרים מהמחלקות הבסיסיות, ואז עליה לספק נתונים אלה בעצמה.

## 5.3.3 ירושה וירטואלית

אם המהירות המקסימלית ברכב אמפיבי היא אחת, ואין הבדל בין מצב ימי ומצב יבשתי, אפשר להגדיר מחלקה נוספת (Vehicle) שתתאר את המושג "רכב" בצורה הכללית ביותר. במקרה זה נגדיר את המחלקות הבאות:

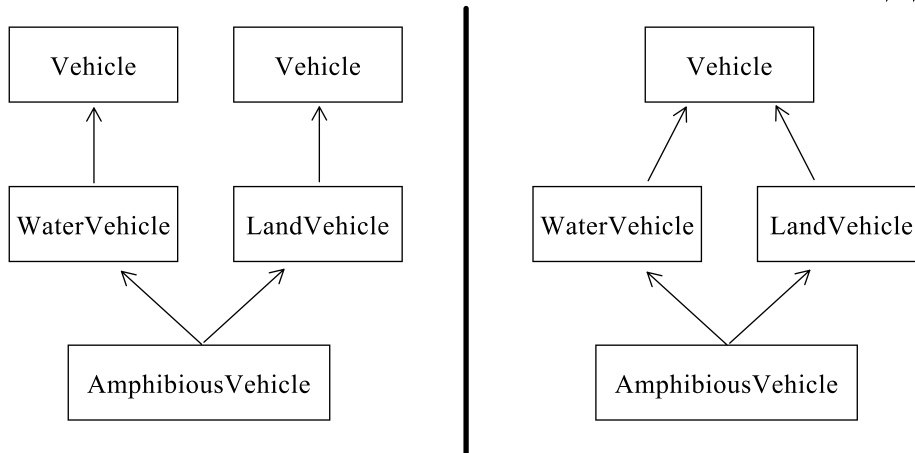
```
class Vehicle {
    int max_speed;
public:
    Vehicle(int s) { max_speed = s; }
    int get_maxspeed(0 const { return max_speed; }
};

class WaterVehicle : public Vehicle {
...
};

class LandVehicle : public Vehicle {
...
};

class AmphibiousVehicle
    : public WaterVehicle, public LandVehicle {
...
};
```

במקרה זה קיבלנו גרף ירושה, שבו לכל אובייקט מסוג רכב אמפיבי נקבל חלק השייך ל-Vehicle פעמיים. כלומר, שתי מהירויות מקסימליות לרכב אמפיבי, למרות שאנו זקוקים לאחת בלבד.




***איור 5.2:** ירושה מרובה רגילה ווירטואלית*

איור 5.2 מתאר בחלקו השמאלי את המצב שתארנו. המחלקות השונות מצוירות כמלבנים שבמרכזם מופיע שם המחלקה. חץ בין שתי מחלקות מסמן יחס ירושה, שכיוונו מהמחלקה היורשת אל המחלקה הבסיסית. בחלקו השמאלי של האיור אנו רואים שלאובייקט מסוג רכב אמפיבי יש שני חלקים המציינים אובייקטים שעוברים בירושה מרכב ימי ומרכב יבשתי. בחלקו הימני של האיור אנו רואים מופע יחיד בגרף הירושה של המחלקה רכב. דבר זה קורה כאשר מגדירים **ירושה וירטואלית**.

כדי לבטל כפילויות כאלו בגרף הירושה, משתמשים בירושה וירטואלית. כאשר נוסיף את מילת המפתח **virtual** ברשימת המחלקות הבסיסיות, לשתי המחלקות WaterVehicle ו-LandVehicle, מצב זה לא יקרה. לכל אובייקט של רכב אמפיבי יהיה רק חלק אחד של רכב. בדרך כתיבה זו, לכל אובייקט רכב אמפיבי תהיה מהירות מקסימלית אחת.

```
class WaterVehicle : virtual public Vehicle {
...
};

class LandVehicle : virtual public Vehicle {

};

class AmphibiousVehicle
      : public WaterVehicle, public LandVehicle {
...
};
```

כדי לפתור מצב זה, נעביר את הירושה לירושה וירטואלית של שתי המחלקות (WaterVehicle, LandVehicle) היורשות מהמחלקה הבסיסית (Vehicle). מחלקה זו (Vehicle) מופיעה יותר מפעם אחת בגרף הירושה. המחלקה הבסיסית שמופיעה מספר כפול של פעמים בגרף הירושה, תופיע עכשיו פעם אחת בלבד. במקרה זה, המחלקה רכב היא **המחלקה הבסיסית הווירטואלית** (virtual base class). פעולה זו פתרה גם את בעיית אי הוודאות וההבחנה בין שדות (או פונקציות) כפולים הנגרמים כתוצאה מירושה מרובה. אפשר להחליף את הסדר בין מילות המפתח virtual ו-public.

## 5.3.3.1 אתחול במצב של ירושה וירטואלית

ראינו שהמחלקה היורשת אחראית לאתחול הבנאי המתאים של המחלקה הבסיסית. דבר זה נכון גם בירושה וירטואלית. אבל כשיש מחלקה אחרת, היורשת מהמחלקה היורשת, אחראית המחלקה היורשת האחרונה לאתחול המחלקה הבסיסית הווירטואלית.

אחד הכללים של ירושה וירטואלית הוא, ש**המחלקה היורשת ביותר** (most derived class) בגרף הירושה אחראית לאתחול המחלקה הבסיסית הווירטואלית. בדוגמה של

היררכיית הרכבים אחראית המחלקה רכב אמפיבי לאתחול של המחלקה רכב, למרות שהמחלקות רכב ימי ויבשתי מאתחלות אף הן את המחלקה הבסיסית. יתרה מזאת, אם המחלקה היורשת הנמוכה ביותר בגרף הירושה לא תאתחל את המחלקה הבסיסית הווירטואלית, נקבל שגיאת הידור כאשר נגדיר, למשל, אובייקטים מסוג רכב ימי או יבשתי. כדי להמחיש חוקים אלה נתבונן בדוגמה הבאה:

```
#include <iostream.h>

class Vehicle {
      int max_speed;
public:
      Vehicle(int ms) { max_speed = ms; }
      int get_max_speed() const { return max_speed; }
};

class WaterVehicle : virtual public Vehicle {
public:
      WaterVehicle(int ms) : Vehicle(ms) {}
};

class LandVehicle : virtual public Vehicle {
public:
      LandVehicle(int ms) : Vehicle(ms) {}
};

class AmphibiousVehicle : public WaterVehicle,
            public LandVehicle {
public:
      AmphibiousVehicle(int ms) : Vehicle(ms),
            LandVehicle(0), WaterVehicle(0)
        { }
};
```

רכב אמפיבי היורש מרכב ימי ורכב יבשתי, מאתחל את הבנאי של רכב, כי הוא המחלקה היורשת ביותר בגרף הירושה. כלומר, זו המחלקה הנמוכה ביותר בגרף הירושה ושממנה לא יורשת מחלקה אחרת. האתחול האחרון נראה אולי מיותר בדוגמה זו, מכיון שהבנאים של מחלקות אלו רק מאתחלים את הבנאי של המחלקה רכב. במקרים רבים אחרים יהיו למחלקות היורשות בדרג הביניים, שדות נוספים שהן מאתחלות.

המהדר מתעלם מהאתחול של המחלקה הבסיסית הווירטואלית (virtual base class) שנעשה במחלקות הביניים במקרה זה, המחלקות רכב ימי ורכב יבשתי, עבור אובייקטים מסוג רכב אמפיבי.

```
inline ostream &operator<<(ostream &out, const Vehicle &v)
{
	return (out << "max_speed=" << v.get_max_speed());
}

int main()
{
	AmphibiousVehicle av(90);
	WaterVehicle wv(50);
	LandVehicle lv(89);
	cout << "Amphibious:" << av
		<< " Water:" << wv
		<< " Land:" << lv << endl;
	return 0;
}
```

התוצאה של התוכנית שלפנינו היא:

```
Amphibious:max_speed=90 Water:max_speed=50
Land:max_speed=89
```

אתחול הבנאי של רכב אמפיבי שולט על ערך המהירות המקסימלית. אולם, כאשר הוגדר אובייקט מסוג רכב ימי, שלט אתחול הבנאי של רכב ימי, וכשהוגדר אובייקט מסוג רכב יבשתי שלט האחרון על ערך המהירות המקסימלית.

## 5.4 סדר בנייה ופירוק של אובייקטים

במקרה של **ירושה מרובה** (multiple inheritance) בין אובייקטים, צריך לשמור על סדר האתחול של תת-האובייקטים המרכיבים אובייקט מורכב. באופן דומה (אך הפוך) צריך לשמור על סדר לפירוק אובייקטים. בסעיף זה נלמד על סדר הבנייה והפירוק של אובייקטים במצב של ירושה.

### 5.4.1 סדר בניית אובייקטים

כאשר יש אובייקט של מחלקה יורשת (למשל, רכב אמפיבי) נבנים תת-האובייקטים שיוצרים אובייקט כזה, מהמחלקה הבסיסית ביותר למחלקה היורשת ביותר. תחילה מופעל **בנאי המחלקה הבסיסית ביותר**, ולאחר מכן הבנאים של המחלקות היורשות. בסוף התהליך מופעל בנאי המחלקה היורשת ביותר.

סדר הפעלת הבנאים ובניית האובייקטים ממחלקה בסיסית למחלקה יורשת נכון, פרט למקרה של ירושה וירטואלית. בירושה וירטואלית נקראים בנאי המחלקות הווירטואליות לפני כל בנאי אחר, ורק לאחר מכן נקראים בנאי המחלקות הבסיסיות. בסוף התהליך מופעלים בנאי המחלקות היורשות.

כאשר מחלקה יורשת ממספר מחלקות, יש סדר קריאה לבנאי המחלקות הבסיסיות. הסדר בין המחלקות הבסיסיות הוא לפי סדר הגדרתן, משמאל לימין. אם כל עניין הבנאים וסידורי הבנייה נראה מסובך, אל דאגה, נראה דוגמה מפורטת בהמשך פרק זה שתבהיר את כל אי ההבנות.

## 5.4.2 סדר פירוק אובייקטים

בעיקרון, סדר פירוק אובייקטים הפוך לסדר בנייתם. תחילה מפורקת **המחלקה היורשת ביותר**, ולאחריו המחלקות הבסיסיות יותר. לפיכך, נקרא תחילה המפרק של המחלקה היורשת ביותר, ולאחריו נקראים המפרקים של המחלקות הבסיסיות יותר.

סדר ההפעלה של המפרקים הוא כזה: תחילה, מפרקים של מחלקות יורשות ולאחר מכן - מפרקים של מחלקות בסיסיות. סדר זה נכון פרט למקרה של ירושה וירטואלית. בירושה וירטואלית ייקראו המפרקים של המחלקות הבסיסיות הווירטואליות (virtual base class) לאחר כל המפרקים האחרים.

כשמדובר בירושה מרובה שבה יש מספר מחלקות בסיסיות לאותה מחלקה, סדר הפירוק של המחלקות הבסיסיות הוא מימין לשמאל על פי הסדר ברשימת המחלקות הבסיסיות. סדר זה הפוך לסדר הבנייה.

## 5.4.3 דוגמה לסדר בנייה ופירוק של אובייקטים

כדי להמחיש את החוקים שתארנו, נוסיף למערכת ההירכיה של רכבים שורש נוסף. נוסיף מחלקה בסיסית שתייצג "אובייקט ימי" (WaterObj) כלשהו. לאובייקט ימי כזה יש תכונה המגדירה באיזה לחץ מקסימלי הוא יכול לעמוד ולאיזה עומק מתחת לפני המים הוא יכול לרדת.

נגזור את המחלקה רכב ימי מהאובייקט הימי הבסיסי. בכל בנאי ומפרק נוסיף הדפסה המתארת את שם הבנאי, או המפרק. כך נוכל לראות מתי נבנה, או מפורק אובייקט, ואת הסדר של האובייקטים השונים.

```
class WaterObj {
    int max_pressure;
public:
    WaterObj(int mp)
    { max_pressure = mp; cout << "WaterObj" << endl; }
    ~WaterObj()
    { cout << "~WaterObj" << endl; }
};

class Vehicle {
    int max_speed;
public:
```

```
    Vehicle(int ms)
    { max_speed = ms;  cout << "Vehicle" << endl; }
    ~Vehicle()
    { cout << "~Vehicle" << endl; }
    int get_max_speed() const { return max_speed; }
};

class WaterVehicle : public WaterObj,
            virtual public Vehicle {
public:
    WaterVehicle(int ms) : WaterObj(55), Vehicle(ms)
    { cout << "WaterVehicle" << endl; }
    ~WaterVehicle()
    { cout << "~WaterVehicle" << endl; }
};

class LandVehicle : virtual public Vehicle {
public:
    LandVehicle(int ms) : Vehicle(ms)
    { cout << "LandVehicle" << endl; }
    ~LandVehicle()
    { cout << "~LandVehicle" << endl; }
};
class AmphibiousVehicle : public WaterVehicle, public
LandVehicle {
public:
    AmphibiousVehicle(int ms) : Vehicle(ms),
            LandVehicle(0), WaterVehicle(0)
    { cout << "AmphibiousVehicle" << endl; }
    ~AmphibiousVehicle()
    { cout << "~AmphibiousVehcile" << endl; }
};
int main()
{
    AmphibiousVehicle av(90);

    return 0;
}
```

פלט תוכנית זו ייראה כך:

```
Vehicle
WaterObj
WaterVehicle
LandVehicle
AmphibiousVehicle
```

```
~AmphibiousVehcile
~LandVehicle
~WaterVehicle
~WaterObj
~Vehicle
```

תחילה נקרא הבנאי של המחלקה "רכב" (Vehicle). דבר זה קורה משום שרכב היא מחלקה בסיסית וירטואלית. ואם לא כן, היה נקרא הבנאי של אובייקט ימי לפניה (כי הוא מופיע ראשון משמאל על רשימת המחלקות הבסיסיות של רכב ימי). כמו כן, ניתן להבחין כי הבנאי של רכב ימי נקרא ראשון ולאחריו הבנאי של רכב יבשתי (שכן האחרון מופיע לימינו של רכב ימי על רשימת המחלקות הבסיסיות של רכב אמפיבי). לבסוף, נקרא בנאי מחלקה הבסיסית ביותר, "רכב אמפיבי".

בפירוק ניתן להבחין שהמפרק של רכב אמפיבי נקרא ראשון (שכן זו המחלקה היורשת ביותר). לאחר מכן, נקרא המפרק של רכב יבשתי (שכן הוא מופיע מימין לרכב ימי ברשימת המחלקות הבסיסיות שמורישות לרכב אמפיבי). אחריו נקרא המפרק של רכב ימי, ואחריו נקרא המפרק של אובייקט ימי. לבסוף, נקרא המפרק של רכב (שהרי רכב היא מחלקה בסיסית וירטואלית). אם המחלקה רכב לא היתה מחלקה בסיסית וירטואלית היה נקרא המפרק שלה לפני המפרק של אובייקט ימי, מפני שהיא מופיעה לימינו של השם האחרון ברשימת המחלקות הבסיסיות של רכב ימי.

## 5.5 המרות

בפרק זה ראינו שהמהדר עושה **המרות** (conversions), כדי להתאים בין סוגים של משתנים לסוגים מבוקשים של משתנים. המרות אלו מתרחשות בקריאות לפונקציות, או בהשמה של משתנים. המרות שונות עשויות לגרום לאובדן מידע, או שיכולה להתבצע פעילות לא צפויה. לכן, חשוב לדעת אילו המרות מתבצעות.

ההמרות מתחלקות ל**המרות סטנדרטיות** (standard conversions) המוגדרות בשפה, ו**המרות המוגדרות על ידי המשתמש** (user defined conversions).

### 5.5.1 המרות של סוגים בסיסיים

בהמרות של סוגים בסיסיים מומרים ערכי משתנים בסיסיים לערכים בסיסיים אחרים. למשל, המרה משלם לתו היא המרה שעלולה לגרום לאובדן מידע, כי מיגוון הערכים של שלמים גדול יותר מתחום הערכים של תו. באופן דומה, המרה של משתנה ממשי למשתנה שלם עלולה לגרום גם היא לאובדן מידע.

```
float x = 5.2;
int j = 303;
char c = j;        // loss of information
j = x;             // loss of info - j <-- 5
```

בשתי ההמרות שעושה המהדר עלול להיות אובדן מידע. הערך של j גדול מערך של תו, ולכן הוא אינו יכול להיכנס למשתנה מסוג char. הערך של c אינו מוגדר לאחר ההשמה. הערך של j לאחר ההשמה הוא 5. במקרה זה יש אובדן מידע.

המרות כאלו נעשות גם כאשר יש קריאות לפונקציה עם ארגומנטים לא מתאימים. לדוגמה, פונקציה המקבלת שלם:

```
int power(int x)
{
      return x * x;
}
float x = power(2.5);
```

הערך 2.5 מומר לערך שלם בעת הקריאה לפונקציה power, והערך המתקבל במשתנה x הוא 4. במקרים כאלה יש מהדרים המוציאים הודעת אזהרה, ואילו אחרים ממירים ללא כל הודעה. המרה מסוג זה היא **המרה בסיסית** (basic conversion) של השפה, או **המרה סטנדרטית**.

## 5.5.2 המרות של מחלקות

המרות אחרות הן בין מחלקות, כאשר אובייקט של מחלקה יורשת יכול להופיע בכל מקום שיש בו התייחסות לאובייקט של מחלקה בסיסית. כשיש שימוש באובייקטים יורשים עבור קוד המטפל באובייקטים בסיסיים, יכולה להיות המרה המאבדת מידע שנקראת **חיתוך אובייקטים** (object slicing), או המרה שאינה מאבדת מידע.

## 5.5.3 חיתוך אובייקטים

חיתוך אובייקטים קורה, למשל, כאשר משימים אובייקט יורש לאובייקט בסיסי. במצב כזה האובייקט היורש מומר לאובייקט בסיסי והתכונות הנוספות שלו אובדות (מכאן המושג "חיתוך"). למשל:

```
class Base {
      int x;
...
};

class Derive : public Base {
      int y;
...
};


Base b;
Derive d;
```

```
b = d;      // object slicing
d = b;      // a compiler error
```

השמה של d ל-b מעבירה לאובייקט הבסיסי רק את התכונות שקיימות בחלק של האובייקט הבסיסי באובייקט היורש d. ההשמה ההפוכה מאובייקט בסיסי לאובייקט יורש אסורה, ותגרום לשגיאת הידור. ההמרות יכולות להיות גם כתוצאה מקריאה לפונקציה. המרות מאובייקט יורש לאובייקט בסיסי הן **המרות סטנדרטיות**.

## 5.5.4 המרות של מצביעים או ייחוס

**המרה של מצביעים** (pointer conversion), או ייחוס ממחלקה יורשת למחלקה בסיסית, נחשבת להמרה סטנדרטית. המרה מסוג זה אינה מאבדת מידע. לדוגמה:

```
Derive d;
Base *bp = &d;

void f(Base *bptr)
{
...
}
void g(Base &br)
{
...
}


f(&d);    // standard conversion to Base pointer
g(d);     // standard conversion to a reference to Base
```

המרת כתובת של מחלקה יורשת למצביע למחלקה בסיסית, נעשית באופן אוטומטי על ידי המהדר. גם המרה של ייחוס לאובייקט יורש לייחוס לאובייקט בסיסי, נעשית בצורה אוטומטית, לכן הקריאות לפונקציות f ו-g הן חוקיות. ההמרה ההפוכה, ממצביע למחלקה בסיסית למחלקה יורשת, אינה נעשית בצורה אוטומטית, והמתכנת צריך לדרוש אותה בפירוש. אם המתכנת אינו משתמש בהמרה מפורשת בקוד ממצביע בסיסי למצביע יורש, יסמן המהדר את שורת הקוד המתאימה כשגיאה.

המרה נוספת, המתבצעת באופן אוטומטי על ידי המהדר, היא המרה ממצביע כלשהו למצביע מסוג void.

## 5.5.5 המרות המוגדרות על ידי המשתמש

**המרות מוגדרות על ידי המשתמש** (User defined conversions), או **המרות משתמש**, מוגדרות על ידי המתכנת להמרה בין סוגי אובייקטים שונים, או בין אובייקטים לסוגים בסיסיים. המרות אלו אינן נחשבות להמרות סטנדרטיות של השפה. הן תוצאה של הפעלת בנאי, או אופרטור המרה. הנה דוגמה:

```
class Y {...};
class X {
...
public:
    X(int x);            // conversion from int to X
    operator int();      // conversion from X to int
    operator Y();        // conversion from X to Y
...
};

void f(X x)
{
...
}

void f(Y y)
{
...
}
int j;
X x;
Y y = x;                 // (1) y = x.Y();
f(j);                    // (2) call to f(X(j));
```

בדוגמה זו, ממיר המהדר את האובייקט X לאובייקט מסוג Y, לצורך שורת ההשמה הראשונה (1). ההמרה נעשית באופן אוטומטי, על ידי שימוש באופרטור ההמרה המתאים של המחלקה X. בשורה השנייה (2) נעשית ההמרה של השלם לצורך התאמה לארגומנט של הפונקציה לאובייקט מסוג X, בשימוש בבנאי המתאים של X.

## 5.5.6 סדר ההמרות

כדי למנוע אי ודאות, מוגדרות **עדיפויות** בין ההמרות השונות. המהדר רשאי לבצע מספר אינסופי של **המרות סטנדרטיות** ו**המרת משתמש** זו המוגדרת על ידו. ההמרות נעשות כדי להתאים משתנים לארגומנטים של פונקציות, או בעת השמות בין משתנים, או ערכים המוחזרים מפונקציות. עדיפות ההמרות מוענקת להמרות סטנדרטיות, ולאחר מכן להמרות המוגדרות על ידי המשתמש.

האלגוריתם המעשי להמרות הוא כזה:

- אם **המשתנים** הם **בסיסיים** (אינם אובייקטים) ולא מתאימים, נסה להמירם לערך שלם.
- המר **מצביעים** למצביע מסוג void.
- המר מצביעים, או ייחוס של **מחלקות יורשות**, למצביעים, או ייחוס, של **מחלקות בסיסיות**.

- השתמש פעם אחת ויחידה **בהמרה שהוגדרה על ידי המשתמש**.

כאשר יש שתי פונקציות, או יותר, המתאימות לקריאה, מועדפות הפונקציות בהן יש פחות המרות ו/או עם המרות סטנדרטיות. לדוגמה:

```
class Y {...};
class X : public Base {
...
public:
    X() {}
    operator Y();        // conversion to Y
};

void f(Base &br);        // (1)
void f(Y y);             // (2)


X x;
f(x);                    // call f(Base &);
```

בדוגמה זו תקבל הפונקציה הראשונה עדיפות, ולכן תקרא הפונקציה המסומנת ב-**(1)**. ההמרה ל-Y מ-X נמצאת בעדיפות נמוכה יותר, ולכן לא תתבצע.

## 5.6 סיכום

בפרק זה למדנו על מושג הירושה ב-++C. ראינו שיש מספר סוגי ירושה ומספר רמות גישה בירושה. יש **ירושה מרובה** ו**ירושה יחידה**, כשלשני סוגי ירושה אלה יש מספר רמות גישה.

רמת **הגישה הפרטית** שקולה להכלת אובייקטים. דהיינו, שקולה להכנסת האובייקטים הבסיסיים בחלקו הפרטי של האובייקט היורש. השדות הציבוריים הופכים לשדות פרטיים, כך שלשדות אלה אין גישה למשתמשים במחלקה היורשת.

ב**ירושה מוגנת** הופכים השדות והפונקציות, המוגנים והציבוריים, למוגנים במחלקה היורשת. לכן, לא יכול המשתמש במחלקה היורשת לגשת לשדות והפונקציות הציבוריות במחלקה הבסיסית.

הפונקציות והשדות הציבוריים של המחלקה הבסיסית ב**ירושה ציבורית**, ציבוריים גם במחלקה היורשת. לכן, יכול המשתמש במחלקה היורשת לגשת לפונקציות של המחלקה הבסיסית.

בירושה מרובה, המחלקה יורשת ממספר מחלקות. הדבר גורם לכפילות של מחלקות בגרף הירושה, וכתוצאה מכך לבעיית הבחנה בין שדות זהים של אותה מחלקה בסיסית המופיעים פעמיים. הפתרון לבעיות מעין אלו ב-++C הוא להגדיר את המחלקה הבסיסית כ**מחלקה וירטואלית בסיסית**, וכך דואג המהדר שהמחלקה הבסיסית תופיע פעם אחת ויחידה בגרף הירושה.

## 5.7 שאלות

1. הוסף מחלקה עובד (Employee) למחלקות Person, Professor ו-Student. דאג לכך שמחלקה זו תירש מ-Person. כמו כן, הכנס את המחלקה עובד כמחלקה בסיסית של פרופסור. הוסף למחלקה הבסיסית עובד שדה המתאר את מספר שעות העבודה החודשיות שלו.
2. הוסף מחלקה נוספת, מזכירה, להיררכיה מהשאלה הראשונה, היכן תוסיף מחלקה **זו** בגרף הירושה?
3. הוסף מחלקה, המייצגת מתרגל, שהיא גם סטודנט וגם עובד, האם יש בעיות של כפילויות בגרף הירושה?
4. הגדר מחסנית בעזרת רשימה בשימוש בפונקציות אחרות מ-insert ו-remove_head.
5. הגדר תור בעזרת פונקציות אחרות מ-append ו-remove_head.
6. הגדר תור דו-כיווני בעזרת הרשימה וירושה פרטית.
7. הגדר רשימה מעגלית מקושרת יחידה.
8. בשימוש ברשימה שהגדרת בשאלה הקודמת הגדר תור ומחסנית. האם השתמשת בירושה ציבורית, מוגנת או פרטית?
9. הגדר איטרטור לרשימה המקושרת הקודמת. מה עליך לעשות כדי להכניס אובייקטים מסוג Person לרשימה זו?
10. הגדר מחלקה Pool המאפשרת לנהל זיכרון בגודל קבוע. למחלקה זו יש פונקציה של הקצאת בלוק, ופונקציה של שחרור בלוק. כשאין למחלקה זיכרון פנוי, היא מקצה מספר רב של בלוקים מהמערכת. מספר הבלוקים המוקצה הוא פרמטר של המחלקה.

פרק 6

# פולימורפיזם

בפרק זה נלמד על המושג פולימורפיזם. **פולימורפיזם** (polymorphism) פירושו **רב-צורתיות**, שמשמעותו: אובייקט מסוים יכול ללבוש צורות רבות. פולימורפיזם ממומש ב-++C בעזרת פונקציות וירטואליות, ולכן מושגים אלה הם שמות נרדפים ב-++C.

המושג פולימורפיזם, בצירוף ירושה ומחלקות, הוא למעשה, התמצית של תכנון ותכנות מוכווני אובייקטים. בשפות רבות נמצא שמחלקות, ירושה ופולימורפיזם הם מקנים להם את ה"זכות" להיות מוגדרות כשפות תכנות התומכות בתכנות מוכוון אובייקטים. ב-++C יש מבנים נוספים, שאינם קשורים לתכנות מוכוון אובייקטים, אך הופכים את השפה לאטרקטיבית מאוד. נבחן נושאים אלה לעומקם בהמשך.

## 6.1 דוגמה - הדרך הישנה

כדי להמחיש את התועלת והחשיבות של המושג **פולימורפיזם**, נפתח פרק זה בדוגמה שאינה משתמשת ביכולת זו.

הדוגמאות לפולימורפיזם מופיעות בקבצים **polym1.cpp**, **polym2.cpp**, **polym3.cpp**. הקובץ הראשון הוא הדוגמה הראשונה והאחרים הם דוגמאות לשימוש בפונקציות וירטואליות, כפי שמוסבר בסעיפים הבאים.

נכתוב תוכנית שתצייר צורות על המסך, ונעשה זאת כמובן בשפת ++C. נגדיר מחלקה Screen שתייצג את המסך ולתוכה נוכל לצייר נקודות. נסביר זאת בהמשך.

נניח, שיש לנו שתי צורות, קו (line) ונקודה (point). פרט לציור הצורות על המסך נרצה לשמור את הצורות בקובץ, כדי שנוכל לחזור ולשנות אותן בעתיד. נחזיק את הנתונים בקובץ במבנה רשימה. כשנרצה לפעול על הצורות, נעבור על הרשימה, נבחר את הצורה הרצויה ונבצע את הפעולות הדרושות.

## 6.1.1 הגדרת המסך

המחלקה Screen תייצג את המסך. זו מחלקה פשוטה מאוד שמחזיקה מערך של תווים המייצג את המסך. בחרתי אמצעי ציור פרימיטיבי כזה, משום שהדגש אינו על הציור עצמו, אלא על הצגת עיקרון עבודה עם מספר צורות והקושי לעשות זאת ללא שימוש בפולימורפיזם.

כאשר רוצים לצייר נקודה, שמה המחלקה Screen תו "*", המייצג נקודה. בהגדרת המחלקה יש enum המייצג את גודל המסך. enum, במקרה זה, מחליף קבועים אותם יכולנו להגדיר בתחום הגלובלי של הקובץ. החיסרון של הגדרה כזו היא בזיהום של מרחב השמות הגלובלי. כאשר הקבועים מוגדרים כ-**enum** בתוך המחלקה, לא קיימת סכנת התנגשויות בין שמות.

```
class Screen {
    enum { nrows = 21, ncols = 77 };
    char data[nrows][ncols];
public:
    Screen() { clear(); }
    void clear();
    void put(int x, int y);
    void draw();
};
```

המערך data מייצג את המסך כמערך כפול. הפונקציה clear ממלאת בתווים ריקים את המערך data שמייצג את המסך. כשמוגדר אובייקט מסוג Screen נקראת פונקציה זו בבנאי האובייקט ומנקה את המסך.

```
void Screen::clear()
{
    for (int r=0; r<nrows; r++)
        for (int c=0; c<ncols; c++)
            data[r][c] = ' ';
}
```

הפונקציה put מכניסה את הערך "*" למקום data, במקום שנתון על ידי הקואורדינטות (x,y). קואורדינטות אלו הן הארגומנטים המועברים לפונקציה. לפני שינוי הערך בנקודה המתאימה במערך נבדק האינדקס, כדי לדעת אם הוא בתוך המערך.

```
void Screen::put(int x, int y)
{   if (x < ncols && x >= 0 && y < nrows && y >= 0)
        data[y][x] = '*';
}
```

כדי לצייר את המסך עוברים על כל השורות ועל כל העמודות במערך ושולחים אותם לפלט הסטנדרטי.

```
void Screen::draw()
{
    for (int r=0; r<nrows; r++) {
        for (int c=0; c<ncols; c++)
            cout << data[r][c];
        cout << endl;
        }
}
```

## 6.1.2 הגדרת הצורות

לצורך הגדרת הצורות נשתמש ברשימה הכפולה, מהפרקים הקודמים. עלינו לגזור את המחלקות שלנו מהצומת שברשימה. נגדיר מספר מחלקות, כדי לאפשר מספר צורות ברשימה.

נגדיר את המחלקה shape כך שתאפשר לנו לגשת לשדה type ולהפיק מכך את הצורה הנוכחית. אם אינך מבין זאת כעת, אל דאגה, הדוגמה תבהיר זאת.

```
enum shape_type { point, line };

struct Shape : public link {
      shape_type type;
};
```

נגדיר שתי מחלקות (Line ,Point) שלהן יש את סוג הצורה בשדה הראשון. כל המחלקות שצריך לצייר, צריכות להגדיר את סוג הצורה בשדה הראשון. נשתמש בשדה זה כדי להבדיל בין סוגי הצורות השונות ברשימה.

המחלקה Point מייצגת נקודה. לכל נקודה יש קואורדינטות המתארות את מיקומה. כדי לצייר נקודה על המסך יש להשתמש בפונקציה put, לפי הקואורדינטות הנתונות שלה.

```
class Point : public link {
      shape_type type;
      int x,y;
public:
      Point(int xo, int yo) : type(point) { x= xo; y=yo; }
      void draw(Screen &scr)
      { scr.put(x,y); }
};
```

המחלקה Line מייצגת קו ישר. יש לה שתי נקודות קצה המייצגות את הקו.

```
class Line : public link {
      shape_type type;
      int x1, y1;
      int x2, y2;
public:
```

```
      Line(int xo1, int yo1, int xo2, int yo2) :
type(line)
      { x1 = xo1; y1 = yo1; x2 = xo2; y2 = yo2; }
      void draw(Screen &scr);
};
```

ציור הקו מסובך יותר מציור נקודה, כי הוא מסתמך על המשוואה הבסיסית של קו במישור:

$$y = a*x + b$$

a מייצג את שיפוע הקו ו-b מייצג את הקבוע שלו. כתוצאה מחישובים אלה אנו עלולים לקבל מספר שאינו שלם, ולכן הפונקציה make_int מעגלת עבורנו מספר ממשי למספר שלם.

```
int make_int(double d)
{
      return int(d+0.5);
}
```

כאשר נתון קו, ייתכן שערך x של שתי נקודות הקצה שלו זהה. במקרה זה, אין שיפוע לקו, ואז מציירים את הקו על ה-x הקבוע בין שתי נקודות y שלו. זהו החלק של else פונקציית הציור של הקו. במקרה השני, מציירים את הקו לפי משוואת הקו.

```
void Line::draw(Screen &scr)
{
     int xmin = (x1 < x2 ? x1 : x2);
     int xmax = (x1 < x2 ? x2 : x1);
     if (x1 != x2) {
         // y = a * x + b
         double a = double(y2 - y1) / double(x2 - x1);
         double b = y1 - a *x1;

         for (int xi = xmin; xi <= xmax; xi++) {
             double y = xi * a + b;
             scr.put(xi, make_int(y));
         }
     }
     else {
         int ymin, ymax;
         if (y1 < y2) {
             ymin = y1;
             ymax= y2;
         }
         else {
             ymin = y2;
             ymax = y1;
         }
```

```
        for (int y = ymin; y<= ymax; y++)
            scr.put(x1, y);
    }
}
```

## 6.1.3 הגדרת הפונקציות והפעולות

מספר פונקציות מטפלות בצורות השונות. הראשונה, היא זו העוברת על רשימת הצורות ומציירת אותן. היא עוברת על רשימת הצורות הנתונה לה, בוחרת את האובייקט לפי סוג הצורה הנתונה, ומבצעת המרה של המצביע לאובייקט המתאים. לאחר ההמרה לאובייקט המתאים, אפשר להשתמש בפונקציית הציור של האובייקט. פונקציה זו חייבת להכיר את כל סוגי הצורות הנמצאים ברשימה. כאשר מוסיפים צורה חדשה, יש לשנות פונקציה זו בהתאם, כך שתטפל בצורה אחרת.

```
void draw(list &slist, Screen &scr)
{
     list_iter i = slist.begin();
     scr.clear();
     while (i != slist.end()) {
          Shape *sp = (Shape*)*i;
          Point *pp;
          Line *lp;
          switch (sp->type) {
               case point:
                    pp = (Point*) sp;
                    pp->draw(scr);
                    break;
               case line:
                    lp = (Line*) sp;
                    lp->draw(scr);
                    break;
               default:
                    break;
          }
          ++i;
     }
     scr.draw();
}
```

בדומה לפונקציה draw, תהיינה לנו פונקציות שכותבות, או קוראות, אובייקטים מקבצים. כל פונקציה כזו צריכה להכיר את כל סוגי האובייקטים, וצריכה להפעיל את הפונקציה המתאימה לאובייקט הנוכחי, לפי סוגו.

הפונקציה הראשית מכניסה שני קווים ושתי נקודות לרשימה, ולאחר מכן מציירת את הצורות הנמצאות בה.

```
int main()
{
    list shapes;
    Screen scr;
    shapes.append(new Line(10,1, 19, 11));
    shapes.append(new Line(40, 20, 55, 10));
    shapes.append(new Point(10, 20));
    shapes.append(new Point(15, 10));
    draw(shapes, scr);
    return 0;
}
```

### 6.1.4 ניתוח חסרונות

בשיטה זו יש מספר פונקציות שעוברות על הרשימה ובוחנות את סוג האובייקט הנוכחי. ציור האובייקט נעשה בדרך שונה על פי סוגו. כשמוסיפים מחלקה חדשה, מלבן לדוגמה, יש לשנות את כל הפונקציות האלו ולהוסיף להן קטע קוד המטפל במחלקה החדשה. חיסרון זה דומה, במידת מה, להסתרת מידע. כשלא משתמשים במודולים ובממשק, שינוי של מבנה הנתונים גורר שינוי של התוכנית כולה. במקרה זה, אנו משתמשים במודולים (שהם המחלקות).

חיסרון אחר הוא, שעלינו לזכור לציין בשדה הראשון בכל צורה את הסוג. כל תוכלנה הפונקציות המטפלות ברשימת הצורות לבצע את עבודתן. אם נשכח לציין זאת, תהיה התוצאה בלתי צפויה ובמקרה הטוב תפסיק התוכנית לעבוד.

שני חסרונות אלה חשובים מאוד מנקודת ראות של הנדסת תוכנה. הם מקשים מאוד על פיתוח ספריית צורות על ידי גורם שלישי. בשינוי קטן מאוד אפשר להתגבר על בעיות אלו ולשפר את איכות התוכנה. הפתרון לכך הוא **פולימורפיזם**, שימוש ברב-צורתיות של השפה.

## 6.2 מהו פולימורפיזם

היה טוב לו יכולנו בדוגמה הקודמת, לבצע פונקציה כלשהי לטיפול בצורה הרצויה, ללא צורך בהמרת הצורה לפי שדה מתאים. היה טוב עוד יותר אם לא היינו צריכים לספק שדה כזה לכל צורה. טוב היה, לו התוכנה היתה מתעלמת מסוגי הצורות השונים.

פולימורפיזם הוא **רב-צורתיות**, כלומר, מצביע או ייחוס לאובייקט יכול להיות מצביע **למספר** סוגים של אובייקטים כאלה. כאשר המצביע מצביע לנקודה מסוימת ואנו מפעילים את הפונקציה draw, היא תופעל על נקודה, ואם המצביע מצביע לקו, היא תופעל על קו. כלומר, הפונקציה draw מזהה את האובייקט ומתאימה את עצמה לפעול עליו בצורה נכונה.

פולימורפיזם וירושה פועלים יחד. יש להגדיר פונקציות אלו במחלקה הבסיסית ואז להגדירם מחדש במחלקות היורשות. כשממלאים תנאים אלה אפשר להשתמש ב**הפעלה וירטואלית** (פולימורפית) של פונקציות.

## 6.2.1 הפעלה וירטואלית

כדי שפונקציה תאפשר פולימורפיזם, יש להוסיף להגדרתה את מילת המפתח **virtual**. כלומר, מגדירים מחלקה בסיסית שיש לה פונקציה וירטואלית. לכל המחלקות היורשות מהמחלקה הבסיסית קיימת אותה פונקציה וירטואלית, אשר "יודעת" לפעול על פי סוג האובייקט. פונקציה וירטואלית מממשת את ההתנהגות הפולימורפית המבוקשת.

כדי להבהיר נקודות אלו נחזור לדוגמה הקודמת, זו עם ציור הנקודה והקו. נגדיר את המחלקה Shape כיורשת מהצומת הבסיסי של הרשימה. למחלקה זו נגדיר את הפונקציה draw כ**פונקציה וירטואלית** (virtual function).

```
class Shape : public link {
public:
      Shape() {}
      virtual void draw(Screen &) {}
};
```

את המחלקות Point ו-Line נגזור כעת מהמחלקה Shape. הפונקציה draw היא וירטואלית במחלקה הבסיסית Shape, לכן, פונקציה זו תהיה וירטואלית גם במחלקות Point ו-Line. הפעלה של פונקציות אלו דרך מצביע למחלקה הבסיסית, תפעיל את הפונקציה המתאימה לאובייקט.

```
class Point : public Shape {
    int x,y;
public:
    Point(int xo, int yo)  { x= xo; y=yo; }
    void draw(Screen &scr)
    { scr.put(x,y); }
};

class Line : public Shape {
    int x1, y1;
    int x2, y2;
public:
    Line(int xo1, int yo1, int xo2, int yo2)
    { x1 = xo1; y1 = yo1; x2 = xo2; y2 = yo2; }
    void draw(Screen &scr);
};
```

השינוי הגדול הוא בפונקציה draw. במקרה זה, אין כל צורך לבדוק את סוג האובייקט, כי הפונקציה המתאימה לאובייקט תופעל, לפי סוגו, על ידי ++C. אם המצביע sp מצביע לאובייקט מסוג point תבוצע פונקציית ציור המתאימה ל-point.

מסקנה נוספת שאפשר להסיק מכך היא, שאם נוסיף סוגים של אובייקטים, כגון מלבן, אין כל צורך לשנות את הפונקציה draw, או פונקציות אחרות שתלויות בסוג האובייקט. מכאן, שאפשר לספק ספרייה שמטפלת בציור של צורות היורשות מ-Shape. כל מי שרוצה להוסיף אובייקט לספרייה, גוזר את האובייקט שלו מהאובייקט הבסיסי Shape, והספרייה תטפל באובייקט ללא צורך במידע מדויק מראש לגבי סוגו.

```
void draw(list &slist, Screen &scr)
{
    list_iter i = slist.begin();
    scr.clear();
    while (i != slist.end()) {
      Shape *sp = (Shape*)*i;
      sp->draw(scr);
      ++i;
    }
    scr.draw();
}
```

פונקציות הכותבות, או קוראות אובייקטים, תהיינה קצרות יותר, ולא תהיינה תלויות בסוג האובייקטים ברשימה.

## 6.3 כיצד מיישמים פונקציות וירטואליות?

לאחר שהבנו את משמעות הפונקציות הווירטואליות (virtual function), ואת היכולת הניתנת לנו כתוצאה מהן, נשאלת השאלה: מה המחיר? האם, כתוצאה מהפעלה של פונקציה וירטואלית, מבצעת המערכת חיפוש כלשהו של הפונקציה? האם, הפעלה של פונקציה כזו דורשת יותר זמן ריצה?

כדי לענות על שאלות אלו יש להבין כיצד מיישמים פונקציות וירטואליות ב-++C. עבור מחלקה בסיסית כמו Shape עם פונקציה וירטואלית, מוסיף המהדר מצביע לטבלה של פונקציות, אשר מכילה מצביעים אל פונקציות וירטואליות.

כשמגדירים מחלקה עם פונקציות וירטואליות מקבלים קוד הדומה לקוד זה:

```
struct Shape {
      void (*vtbl[])();
};
```

vtbl הוא מצביע למערך של מצביעים לפונקציות. כשצריך להפעיל פונקציה וירטואלית, המהדר מפעיל את הפונקציה באמצעות המצביע לתחילת הטבלה של הפונקציות וההיסט בטבלה זו. כך, הפקודה

```
sp->draw(scr);
```

שקולה לפקודה זו:

```
(*vtbl[1])(sp, scr);
```

תחילה מופעלת הפונקציה הראשונה בטבלה של הפונקציות הווירטואליות ומועבר אליה מצביע לאובייקט הנוכחי, ומצביע this לאובייקט מסוג Screen. הפעלה כזו מהירה ואינה צורכת זמן רב.

טבלת הפונקציות הווירטואלית מאותחלת עבור כל אובייקט. עבור המחלקה Point יוצר המהדר טבלה שבה המצביע לפונקציה הראשונה המתאים ל-draw מצביע אל הפונקציה של המחלקה Point. בדומה לכך, מאותחלת הטבלה להצביע לפונקציה המתאימה בשאר הכניסות שלה. למחלקה Line יש טבלה דומה שמצביעה על הפונקציות של Line.

אתחול המצביע עם אובייקט לטבלה המתאימה נעשה בקוד שמוסיף המהדר בבנאי של האובייקט. אם מנסים להפעיל פונקציה וירטואלית בבנאי של אובייקט בסיסי, תופעל הפונקציה של המחלקה הבסיסית, ולא של המחלקה האמיתית של האובייקט:

```
class Base {
public:
    Base() { print(); }
    virtual void print() { cout << "Base" << endl; }
};

class Derive : public Base {
public:
    Derive() { print(); }
    virtual void print() { cout << "Derive" << endl; }
};

Derive d;
```

התוכנית הזו תדפיס את השורות הבאות:

```
Base
Derive
```

למרות שהאובייקט מסוג Derive מצוי בבנאי של Base, תופעל הפונקציה print, השייכת למחלקה Base. הסיבה היא, שהמצביע לטבלת הפונקציות הווירטואליות מצביע באותה עת לטבלה של Base. רק לאחר שנבנה האובייקט במלואו (עד למחלקה היורשת ביותר) יכול המצביע לטבלה של הפונקציות הווירטואליות להצביע על הטבלה הנכונה.

## 6.4 אילו פונקציות יכולות להיות וירטואליות?

כל הפונקציות יכולות להיות פונקציות וירטואליות פרט לבנאי המחלקה. מכאן נובע, שגם פונקציית הפירוק של המחלקה יכולה להיות פונקציה וירטואלית. המשמעות של מפרק וירטואלי היא, שכשמפעילים את האופרטור delete על מצביע של המחלקה הבסיסית, יופעל המפרק המתאים לאובייקט. דהיינו, יופעל המפרק של המחלקה שאליו שייך האובייקט. לדוגמה:

```
class Shape {
public:
      Shape() {}
      ~Shape() {}
      virtual void draw(...) {}
};

class TextObj : public Shape {
      char *str;
public:
      TextObj(const char *s)
      { str = new char[strlen(s) + 1]; strcpy(str, s); }
      ~TextObj() { delete [] str; }
      void draw(...) { ...}
};
```

במקרה זה, כאשר מפעילים את המפרק של Shape באמצעות קריאה לאופרטור delete למחיקת מצביע מסוג Shape, לא יופעל המפרק של TextObj, אלא המפרק של Shape בלבד. נתבונן בדוגמה הבאה:

```
Shape *sp = new TextObj("test");
delete sp;
```

כאן נאבד זיכרון שהוקצה בבנאי של TextObj, כי המפרק של אובייקט זה לא יופעל. כדי להתגבר על בעיה זו עלינו להגדיר **מפרק וירטואלי** (virtual destructor) למחלקה Shape.

```
class Shape {
...
      virtual ~Shape() {}
};
```

כאשר המפרק הוא וירטואלי, הפעלת אופרטור delete על מצביע ל-Shape תפעיל את המפרק המתאים לאותו אובייקט ולמעשה, את המפרק של האובייקט היורש. לאחר שיופעל המפרק של האובייקט היורש יופעל המפרק של האובייקט הבסיסי, ובעיית שחרור הזיכרון תיפתר.

## 6.5 מחלקות מופשטות טהורות

**מחלקות מופשטות טהורות** (Pure Abstract Class) הן מחלקות שיש להן פונקציה וירטואלית אחת או יותר ללא הגדרה. כלומר, אין גוף לפונקציה. פונקציה וירטואלית ללא הגדרה נקראת לעיתים: **פונקציה וירטואלית טהורה** (Pure Virtual Function). פונקציה כזו מוגדרת כשווה לאפס:

```
class Shape {
public:
    Shape() {}
    virtual ~Shape() {}
    virtual void draw(Screen &) = 0;
};
```

הפונקציה draw היא פונקציה וירטואלית טהורה. מחלקה כזו, שהיא **מחלקה מופשטת**, אינה מאפשרת להגדיר אובייקטים ואין לה מופעים. הסיבה לכך היא, שחסרה לה פונקציה. המהדר אינו מאפשר להגדיר אובייקטים של מחלקה. אפשר להגדיר אובייקטים של מחלקות יורשות ממחלקה זו, בתנאי שהן אינן מופשטות בעצמן.

**מחלקה יורשת** יכולה להיות מופשטת כאשר היא אינה מגדירה את הפונקציה הווירטואלית הטהורה של המחלקה הבסיסית כאשר היא מחלקה מופשטת, או שהיא בעצמה מגדירה פונקציה וירטואלית טהורה.

אם כן, מדוע להגדיר מחלקות כאלו? קיימות מספר סיבות לכך וננתח אותן כעת. למחלקה Shape יש פונקציה draw עם גוף ריק מתוכן, כלומר גוף הפונקציה אינו קיים. הפונקציה מגדירה ממשק בו יש להשתמש כשרוצים לקרוא לפונקציות כאלו. מחלקה מופשטת מגדירה לנו ממשק לרמה היררכית, שכל המחלקות היורשות ממנה מקימות את הממשק הזה.

סיבה נוספת להגדרה של מחלקה מופשטת טהורה היא, להכריח את המחלקה היורשת להגדיר את הפונקציות הנדרשות. המהדר לא יאפשר להגדיר אובייקטים של מחלקות מופשטות טהורות ולכן המתכנת הכותב את המחלקה היורשת, יהיה חייב להגדיר את הפונקציות המתאימות, אם ברצונו להגדיר או ליצור אובייקטים ממחלקה זו.

**מחלקה מופשטת טהורה** מגדירה, אם כן, ממשק שבו משתמשת התוכנה כדי לבצע פעולות על סוגי אובייקטים שאינם ידועים מראש.

קודם לכן ראינו שכשמפעילים פונקציה וירטואלית של אובייקט יורש בבנאי של אובייקט בסיסי, נקבל הפעלה של פונקציה וירטואלית של האובייקט הבסיסי, מכיון שהאובייקט בנוי חלקית, באותה עת. כאשר נפעיל פונקציה וירטואלית טהורה בבנאי של מחלקה בסיסית, נראה שבמקרה הטוב תופסק התוכנית ובמקרה הפחות טוב, ההתנהגות אינה מוגדרת ואינה צפויה.

## 6.5.1 נקודת התורפה של מערכת פולימורפית

נקודת התורפה של מערכת פולימורפית היא **הגדרת האובייקטים**. פרט למקום בו אנו יוצרים את האובייקטים, איננו צריכים לדעת את הסוג, או את המחלקה האמיתית של האובייקט. נקודות היצירה של האובייקטים הן הנקודות ש"מקלקלות" תוכנה הכתובה בדרך זו.

נניח, למשל, שיש לנו ספריית אובייקטים המטפלת בממשק למשתמש. ספרייה זו יכולה לעבוד במספר סביבות שונות, למשל בסביבת Motif תחת X-Windows או בסביבת Windows של מיקרוסופט. עוד נניח, שיש אובייקטים, כגון Button ו-List. לחצן (Button) הוא אלמנט גרפי, שכשבוחרים בו בעזרת העכבר מבצעים פעולה. רשימה (List) היא אלמנט גרפי המציג מספר אפשרויות, ומאפשר לבחור באחת מהן.

```
class MotifButton {
...
      void select();
};

class MotifList {
...
      void select();
};

class MSButton {
...
      void select();
};

class MSList {


...
      void select();
};
```

אובייקטים גרפיים אלה מספקים פונקציונליות זהה בשתי סביבות העבודה. כאשר נבחר הלחצן נקראת הפונקציה select, ומכיון שהתוכנה שלנו פועלת בשתי סביבות אלו, עליה להשתמש בפונקציה כזו. אם בכל מקום בתוכנה ניצור אובייקטים גרפיים כאלה, יהיה קשה לתחזק את התוכנה בשתי סביבות העבודה ובסביבות אחרות יגרום הדבר לבעיית תחזוקה קשה מאוד. לדוגמה:

```
#ifdef MOTIF
   MotifButton mb(...);
#else
   MSButton msb(...);
#endif
```

### 6.5.1.1 ייצור אובייקטים מופשטים (Abstract Factory)

קטעי הקוד שתוארו כאן תלויים בסוג האובייקטים. רצוי להסיר תלות זו. כיצד נתגבר על הבעיה? נעשה זאת על ידי הגדרה של מחלקות מופשטות עבור לחצנים ורשימות, כך:

```
class Button {
...
   virtual void select() = 0;
};

class List {
...
   virtual void select() = 0;
};
```

מחלקות מופשטות אלו מגדירות את הפונקציות שהתוכנה מכירה. בכל מקום בתוכנה שלנו נשתמש במצביעים ללחצנים, או לרשימות גרפיות. ניצור את המחלקות הבאות:

```
// Motif buttons and list
class MotifButton : public Button {
...
   void select();
};

class MotifList : public List {
...
   void select();
};

// Microsoft windows button and list
class MSButton : public Button {
...
   void select();
};

class MSList : public List {
...
   void select();
};
```

אם כל התוכנה שלנו משתמשת במצביעים למחלקות המופשטות, אין צורך לשנות דבר כשעוברים מסביבת עבודה אחת לאחרת. כל זה נכון, פרט לנקודה של הגדרת האובייקטים. כדי שמצביעים אלה יקבלו ערכים נכונים, יש לתת להם כתובת של אובייקטים. לכן יש להכיר את האובייקטים של הסביבה המתאימה.

כדי למנוע הכרת סוג אובייקטים בכל מקום בסביבת העבודה, ניצור עוד מחלקה מופשטת היוצרת לחצנים ורשימות בצורה הבאה:

```
class Creator {
...
   virtual Button *createButton() = 0;
   virtual List *createList() = 0;
};
```

לכל סביבה ניצור מחלקה, היורשת ממחלקה מופשטת זו, ויוצרת את האובייקטים המתאימים לסביבת העבודה. המחלקה תוגדר כך:

```
class MotifCreator : public Creator {
...
  virtual Button *createButton()
      { return new MotifButtton(); }
  virtual List  *createList()  { return new MotifList(); }
};

class MSCreator : public Creator {
...
  virtual Button *createButton() { return new MSButton(); }
  virtual List   *createList()   { return new MSList(); }
};
```

בכל מקום בו נרצה ליצור אובייקטים חדשים, נשתמש במצביע לאובייקט שיוצר אובייקטים:

```
Creator *creator;
...
Button *btn = craetor->creatButton();
```

כעת, נגדיר את סוג יוצר האובייקטים במקום אחד בתוכנה בלבד:

```
creator = new MotifCreator;
```

בדרך זו, חסינה כל המערכת לשינוי סביבת העבודה. אם רוצים לעבור לסביבת עבודה אחרת, נותר רק לשנות את יוצר האובייקטים בדרך הבאה:

```
creator = new MSCreator;
```

השימוש במחלקות מופשטות, יוצר אם כן, תוכנה באיכות גבוהה יותר וחסינה בפני שינויים גדולים, ואפילו שינוי בסביבת העבודה.

## 6.6 סיכום

בפרק זה למדנו את המונח **פולימורפיזם**. ב-++C מאפשר מונח זה **הפעלה וירטואלית** של פונקציות. הפעלה וירטואלית של פונקציות נעשית באמצעות מצביעים, או משתנים, מסוג ייחוס לאובייקטים מסוג בסיסי. הפעלה וירטואלית של פונקציות פועלת בעזרת ירושה. כשיש מצביע למחלקה בסיסית, גורמת הפעלה של פונקציה וירטואלית דרך מצביע כזה, להפעלה של הפונקציה המתאימה לסוג האובייקט הנוכחי.

פולימורפיזם מאפשר, אם כן, להגדיר פונקציות גנריות המטפלות במצביעים למחלקות בסיסיות בלבד. פונקציות אלו אינן מושפעות, למשל, כאשר מוסיפים סוגי אובייקטים חדשים למערכת.

נקודת התורפה היחידה של מערכת פולימורפית היא במקום יצירת האובייקטים. במקום זה יש לכתוב את סוג האובייקט האמיתי, ולא את סוג המחלקה הבסיסית. יש להימנע מפיזור יצירת אובייקטים ולרכזם במקום יחיד, למשל במחלקה אחת.

מחלקה מופשטת טהורה היא מחלקה המגדירה ממשק בלבד, ויש לה פונקציה וירטואלית טהורה אחת או יותר. **פונקציה וירטואלית טהורה** היא פונקציה המסומנת כשווה לאפס בהגדרת המחלקה, ואין לה גוף.

## 6.7 שאלות

1. כתוב פונקציה הכותבת צורות ומחלקות היורשות מהמחלקה צורה לקובץ. היעזר במנגנון הפולימורפי לשם כך. אילו פונקציות עליך להוסיף ל-Shape? האם אלו פונקציות וירטואליות טהורות?
2. כתוב את הפונקציות הקוראות אובייקטים היורשים מהמחלקה Shape מקובץ. מה ההבדל בין פונקציות כותבות לקוראות? כיצד תזהה את סוג האובייקטים שצריך ליצור? האם המערכת שלך עמידה בפני תוספת של מחלקות?
3. הרשימה list מכריחה את המשתמש בה לרשת מהצומת הבסיסי link. האם יש דרך לתקן מצב זה?
4. כתוב ספרייה המאפשרת לקרוא ולכתוב אובייקטים כלשהם מקובץ (רמז: השתמש בפונקציות וירטואליות ומחלקות מופשטות כדי להגדיר את הממשק לאובייקט בסיסי של הספרייה. כל אובייקט שצריך להישמר לקובץ צריך לרשת מהאובייקט הבסיסי של הספרייה).
5. כתוב מחלקה (או מספר מחלקות) אשר אי-אפשר לרשת ממנה (רמז: היעזר בירושה וירטואלית).

# פרק 7

# תבניות ב-C++

בפרק זה נלמד את המונח **תבניות** ב-C++ (templates). תבניות מאפשרות **תכנות גנרי** (generic programming) בדומה לירושה ופונקציות וירטואליות. ובכל זאת, קיימים הבדלים בין **ירושה ופולימורפיזם** לבין **תבניות**. יש דברים רבים שניתן לממש בעזרת כל אחד מכלים אלה. לעומת זאת, יש דברים הניתנים לביצוע באחד מהם בלבד.

נוסף לפונקציונליות שונה של ירושה ופולימורפיזם לעומת תבניות, יש הבדל ביעילות של המערכת הממומשת בשיטות אלו. נבחן עניינים אלה ואחרים בפרק זה.

## 7.1 לשם מה תבניות

לפני שנתחיל ללמוד את הסינטקס והסמנטיקה של **תבניות** (templates), נבין מהן תבניות ומה השימוש שנעשה בהן. נניח, למשל, שיש לנו את המאקרו הבא, שדומה לפונקציה:

```
#define min(x, y)  (x < y ? x : y)
```

כבר ראינו שיש בעיות העלולות להיגרם כתוצאה משימוש במאקרו. למשל:

```
int x,y;
...
if (min(x++,y) < 3) {
...
}
```

קטע קוד זה עלול להגדיל את X פעמיים, למרות שהמתכנת התכוון להגדילו פעם אחת בלבד. הצגנו את הפתרון הבא לבעיה זו:

```
inline int min(int x, int y) { return (x<y ? x : y); }
```

ואכן, כאשר העברנו את המאקרו לפונקציה מסוג זה נפתרת הבעיה, ועתה X יוגדל פעם אחת בלבד. אבל יש לנו בעיה אחרת, למשל:

```
float x=5.1, y=5.2;
float z = min(x, y);
```

הערך שיקבל Z הוא 5. הסיבה לכך היא, שהמהדר ממיר את הפרמטרים של הפונקציה לשלמים וכתוצאה מכך אובד הדיוק. במהדרים רבים נקבל על כך אזהרה, הקוד יעבור את שלב ההידור ונוכל להריץ את התוכנית. במהדרים אחרים, לעומת זאת, לא נקבל את האזהרה ותהיה לנו בעיה בזמן ריצה.

כדי לפתור בעיה זו נצטרך להוסיף את הפונקציה הבאה:

```
inline float min(float x, float y)
{ return (x<y ? x : y); }
```

כעת, נפתרה הבעיה עבור משתנים מסוג float, אך עבור משתנים מסוג double הבעיה עדיין קיימת. האם עלינו ליצור פונקציה כזו עבור כל סוג משתנה? טוב היה לו יכולנו להגדיר את הפונקציה באופן כללי, ולהשאיר למהדר את יצירת הפונקציות המתאימות, כדי שיקרא לפי הצורך, כלומר, לפי סיווג הארגומנטים בעת הקריאה לפונקציה. זה בדיוק מה שנשיג אם נשתמש ב**תבנית** ב-++C.

דוגמה אחרת לכך היא הגדרה של מחלקות. המחלקה list מהפרקים הקודמים אינה צריכה "לדעת" על האובייקטים הנשמרים בצמתים (האיברים) שלה, כדי לממש את הפונקציונליות שלה. היו לנו קטעי קוד כגון:

```
void display(list &lst)
{
    list_iter i(lst.begin());
    while (i != lst.end()) {
      Shape *sp = (Shape*)*i++;

      sp->draw(...);
    }
}
```

כשרצינו להשתמש ברשימה, היה עלינו לגזור את האובייקטים המבוקשים מהצומת הבסיסי של הרשימה, ולבצע את המרות המצביעים. רצוי להימנע מפעולות מיותרות כאלו, והיה טוב אם המהדר היה עושה זה עבורנו, באופן אוטומטי ללא טעויות.

## 7.2 תבניות ב-++C

כפי שראינו בסעיף הקודם, היו מצבים בהם היינו צריכים תבניות עבור פונקציות ותבניות עבור מחלקות. ++C מאפשרת זאת. בהמשך הפרק נתאר זאת.

### 7.2.1 פונקציות תבנית

++C מאפשרת לנו להגדיר **פונקציות תבנית** (template functions). פונקציית תבנית אחת מייצגת **משפחה** של פונקציות שהמהדר מייצר בעצמו על פי פונקציית התבנית. בסעיף הקודם ראינו מאקרו (או פונקציות) ששימשו למציאת המספר המינימלי בין

שני מספרים נתונים. ראינו, שהיה עלינו להגדיר מספר רב של פונקציות לכל סוג של מספרים (שלמים, ממשיים וכו'). אפשר להגדיר פונקציית תבנית ב-C++ בצורה הבאה:

```
template <class T>
T min(T x1, T x2)
{
      return (x1 < x2 ? x1 : x2);
}
```

הגדרה מעין זו עדיין לא ראינו. בהגדרה זו יש מילת מפתח חדשה - **template**, אשר מציינת שלפנינו **פונקציית תבנית** (template function). לאחר מכן, מופיעים הסימנים "< >", שביניהם מופיעים הפרמטרים של ה**תבנית**. כל פרמטר מופיע בפורמט הבא:

```
class T
```

כלומר, מילת המפתח ולאחריה שם הפרמטר. במקרה זה יש רק פרמטר אחד. השמות ברשימת הפרמטרים חייבים להופיע לפחות פעם אחת ברשימת הפרמטרים של הפונקציה שמשתמשת בהם. במקרה זה, השמות של הסוג T מופיעים פעמיים, כסוג של הארגומנטים של הפונקציה x1 ו-x2. הפרמטרים ברשימת הפרמטרים של פונקציית התבנית משמשים כסוגים עבור הארגומנטים של הפונקציה.

הגדרת פונקציה כזו **אינה יוצרת** שום קוד. ההגדרה יוצרת עבור המהדר **תבנית** שממנה הוא יוכל ליצור משפחה שלמה של פונקציות min בהתאם לשימוש בהן. כאשר נקראת פונקציה זו עם משתנים מסוג שלם (int), ייצר המהדר פונקציה המתאימה לשלמים. כאשר ישתמשו בפונקציה עם פרמטרים מסוג float, ייצר המהדר פונקציה מתאימה לסוג זה. פונקציות אלו זהות לפונקציות שהיינו כותבים בנפרד לכל סוג משתנה, כפי שעשינו בסעיף הקודם.

התבנית שכתבנו כאן עונה על שתי הבעיות שהצגנו בסעיף הקודם. ראשית, אין צורך לכתוב פונקציה כזו עבור כל סוג בסיסי. שנית, הסמנטיקה של פונקציית תבנית זהה לסמנטיקה של פונקציה רגילה. לכן, בקטעי קוד מהסוג -

```
int  x,y;
if (min(x++, y))
{ ...}
```

תבוצע ההגדלה של המשתנה x פעם אחת בלבד, ללא כל תלות אם x גדול מ-y.

### 7.2.1.1 שימוש במספר פרמטרים בתבנית

למרות כל האמור לעיל, עדיין יש יתרון למאקרו על פונקציות כאלה (זכור, למאקרו יש גם מספר חסרונות). למשל:

```
#define min(x, y) ((x) < (y) ? (x) : (y))

double x,z;
int y;
z = min(x,y);
```

שורות קוד כאלו יעברו הידור. לעומת זאת, אותה שורת קוד בשימוש בפונקציית התבנית, לא תעבור הידור ברוב המהדרים, מכיון שברשימת הפרמטרים של הפונקציה min יש פרמטר אחד בלבד מסוג T, ואנו מעבירים לפונקציה פרמטרים משני סוגים, שלמים וממשים. כדי להתגבר על כך, נוסיף עוד פרמטר לרשימת הפרמטרים:

```
template <class T1, class T2>
T? min(T1 x1, T2 x2)
{
    if (x1 > x2)
       return x1;
    else
       return x2;
}
```

הבעיה כאן היא, שאין אנו יודעים מהו סוג הפרמטר שיש להחזיר בקריאה מהפונקציה. לרוע המזל, C++ אינה מאפשרת להכניס את הערך המוחזר של הפונקציה כפרמטר של תבנית. הסיבה לכך נעוצה בעובדה שהקוד הקורא לפונקציה אינו חייב להשתמש בערך המוחזר. לכן, אין אפשרות למהדר להבחין באיזו פונקציה מדובר. למשל:

```
template <class T1, class T2>
T2 f(T1 val)
{
//...
}

template <class T1, class T2>
T1 f(T2 val)
{
//...
}

int v;
f(v);  // Which version to choose ?
```

בדוגמה זו אין למהדר כל דרך לבחור את הפונקציה המתאימה. לכן, חייבים להוסיף פרמטר שמתאר את הערך המוחזר של הפונקציה בארגומנטים של הפונקציה. לאחר הוספת הפרמטר תיראה הפונקציה min כך:

```
template <class V1, class V2, class Retval>
void min(V1 v1, V2 v2, Retval &rv)
{
      if (v1 < v2)
            rv = v1;
      else
            rv = v2;
}
```

בצורה זו מועבר הערך המוחזר בפונקציה **כייחוס** (reference) לפונקציה. לפיכך, השימוש בפונקציה הוא כזה:

```
int x;
doubel y, v;

min(x, y, v);
```

התוצאה מניבה קוד פחות קריא מאשר הקוד עם פקודת מאקרו. לעומת זאת, השימוש במאקרו גורם למספר בעיות בתוכנית, כגון סמנטיקה שאינה מוגדרת. אין אפשרות לעבור עם debugger על מאקרו. המרות למשתנים של מאקרו גורמות לאבדן דיוק, או שגיאות ללא כל דיווח של המהדר.

טכניקה אחרת לפונקציה כמו min מציעה שימוש ריק של פרמטר ההחזרה של הפונקציה. בדרך זו:

```
template <class T1, class T2, class Retval>
Retval min(T1 v1, T2 v2, Retval*)
{
    if (v1 < v2)
       return Retval(v1);
    return Retval(v2);
}
```

במקרה זה אנו כוללים ברשימת הארגומנטים של הפונקציה את המצביע לסוג האובייקט שרוצים להחזיר. השימוש ריק, כלומר אין לנו משתנה כזה, וכל תפקידו של הארגומנט לאפשר למהדר לבחור בפונקציה המתאימה. השימוש בפונקציה min הוא כזה:

```
int x;
float y;
double z;

z = min(x, y, (double*)0);
```

היתרון של גישה זו הוא, שכעת אפשר לקבוע סוג פרמטר מוחזר מהפונקציה שאינו קשור לשני הראשונים. מצב זה לא קורה במאקרו, כי סוג הערך המוחזר הוא אחד משני הפרמטרים המסופקים למאקרו.

החיסרון העיקרי בפונקציות שראינו עד כה הוא שהשתמשנו באופרטור הבסיסי "<" של השפה. אם נשתמש בפונקציות אלה עבור *char לא נקבל סדר לקסיקוגרפי (אלפביתי), אלא לפי ערכי המצביעים, דבר שאינו רצוי במקרים אלה. כלומר:

```
char *s2 = "abc", *s1 = "bcd", *s3;
s3 = min(s1, s2);
```

כלל לא מובטח ש-s3 יצביע למחרוזת "abc". אם מחרוזת זו ממוקמת בכתובת שאחרי המחרוזת השנייה, יצביע s3 למחרוזת השנייה (s2). בהמשך נראה כיצד מתמודדים עם בעיות מסוג זה.

דוגמה אחרת של פונקציית תבנית היא הפונקציה swap. פונקציה זו מחליפה בין שני אובייקטים. זו פונקציה גנרית שיכולה להחליף כל שני אובייקטים. היא מקבלת שני ייחוסים לאובייקטים מסוג T, שהם הפרמטרים של התבנית, ומחליפה את הערכים ביניהם.

```
template <class T>
void swap(T &v1, T &v2)
{
    T tmp(v2);
    v2 = v1;
    v1 = tmp;
}
```

הפונקציה swap שימושית לאלגוריתמים נוספים שנלמד בפרק זה.

## 7.2.1.2 תכנות גנרי

**תכנות גנרי** (generic programming) בפונקציות תבנית היא טכניקה המאפשרת לכתוב פונקציות גנריות הטובות למספר רב של יישומים. הפונקציה min היא פונקציה כזו. בסעיף זה נראה דוגמה מורכבת יותר של תכנות גנרי, תוך שימוש בפונקציות תבנית. הדוגמאות הנתונות דומות מאוד לקוד שמומש בספריית האובייקטים הסטנדרטית של C++, שנסקור אותה בהמשך.

בתכנות גנרי משתמשים בתבניות כדי להגדיר אלגוריתמים שיהיו טובים ליישומים רבים ולסוגים רבים של משתנים ואובייקטים. סוג המשתנה, או האובייקט, מיוצג על ידי פרמטר של פונקציית התבנית (או של מחלקת התבנית). כשמשתמשים בפונקציה עבור אובייקטים מסוגים שונים, או עבור משתנים שונים, יוצר המהדר את הפונקציות המתאימות לסוג שבו אנו עוסקים.

בסעיף זה נראה מספר דוגמאות לפונקציות גנריות, ונלמד כיצד יש לתכנן אלגוריתמים כאלה. נעמוד על איכות האלגוריתמים ונסקור את משמעות סיבוכיות האלגוריתמים.

### 7.2.1.2.1 סיבוכיות האלגוריתמים

להערכת איכות של אלגוריתם נתון יש שני מדדים.

- **סיבוכיות זמן** (time complexity) - כמות הזמן שצורך אלגוריתם מסוים להפקת פלט בגודל נתון.
- **סיבוכיות מקום** (space complexity) - כמות הזיכרון שצורך האלגוריתם עבור פלט מסוים.

#### סיבוכיות זמן

**סיבוכיות זמן** (time complexity) מציינת את משך הזמן לפתרון אלגוריתם, והיא נמדדת באופן יחסי למספר האובייקטים, או לגודל הפלט שמקבל האלגוריתם.

לדוגמה, סיבוכיות הזמן של אלגוריתם מיון שמקבל 100 אובייקטים, למשל, נמדדת באופן יחסי למספר האובייקטים. נסמן את מספר האובייקטים בקלט של אלגוריתם באות n. נשתמש באות T (מלשון Time) לציון סיבוכיות הזמן.

במקרה של פונקציה שסורקת מערך ומדפיסה את האלמנטים הנמצאים בו, נאמר שסיבוכיות הזמן שלה יחסית למספר האובייקטים. כך אנו מגיעים למונח **O** (אות "או", ולכן אומרים: **או גדול**).

המונח **או גדול** מציין יחסיות. ההגדרה המדויקת של המונח **או גדול** היא: עבור פונקציה f כזו אומרים שהיא **או גדול** של פולינום אחר g, אם קיימים N ו-K כך שעבור n>N מתקיים:

```
f(n) < K*g(n) = O(g(n))
```

פונקציה המדפיסה אלמנטים שנמצאים במערך תעבוד בסיבוכיות זמן של O(n), כאשר n הוא מספר האובייקטים במערך. זאת משום שאנו יכולים למצוא קבוע K מספיק גדול, כך שבזמן הפעולה של הפונקציה T(n) מקיימים את המשוואה הבאה:

```
T(n) <= K*n + b
```

כש-b הוא קבוע כלשהו. במקרה זה, סיבוכיות הזמן פרופורציונלית למספר האובייקטים במערך. דוגמה אחרת, היא אלגוריתם הממיין מערך של אובייקטים. אם קיים אלגוריתם הממיין את המערך בצורה יחסית ל-n בריבוע, הוא איטי בהרבה מאלגוריתם הממיין את המערך בסיבוכיות זמן O(n*logn). הסיבה לכך היא שהפונקציה logn גדלה לאט יותר מ-n.

אלגוריתם הפועל בזמן קבוע O(k), הוא אלגוריתם שזמן הביצוע שלו קבוע ואינו תלוי במספר האובייקטים בקלט שלו. בדרך כלל, אין אלגוריתמים כאלה.

אלגוריתם לחיפוש אובייקט באוסף כלשהו בצורה O(n), איטי יחסית לאלגוריתם שמוצא אובייקט בסיבוכיות זמן O(logn). אם, לדוגמה, יש באוסף 1000 אובייקטים, תעשינה באלגוריתם הראשון כ-1000 פעולות. באלגוריתם השני, לעומת זאת, תעשינה כ-10 פעולות, לכל היותר.

אלגוריתמים הצורכים זמן רב יותר הם מסדר גודל של O(n*n). בהמשך נראה אלגוריתמים בסדר גודל נמוך, בדרך כלל. נשתדל שלא לכתוב אלגוריתמים עם סיבוכיות זמן כזו, כלומר - O(n*n).

### סיבוכיות מקום

**סיבוכיות מקום** (space complexity) תלויה גם היא במספר האובייקטים שבקלט. סיבוכיות המקום היא כמות הזיכרון שהאלגוריתם צורך. במקרה זה, נסמן את סיבוכיות המקום באות הפונקציונלית S. כמו סיבוכיות הזמן, כך גם סיבוכיות המקום ניתנת לתיאור בעזרת הפונקציה O. האלגוריתמים שנתאר כאן יהיו בעלי סיבוכיות מקום קבועה, בדרך כלל.

כדי להדגים עקרונות אלה, נראה עתה מספר פונקציות גנריות המממשות מספר אלגוריתמים, כמו גם תוכנית קטנה המשתמשת בהן.

### 7.2.1.2.2 דוגמה - אלגוריתם הפיכת סדר

הדוגמאות בפרק זה נמצאות במספר קבצים שבדיסקט המצורף:

- קובץ **algo.cpp** מכיל את האלגוריתמים הגנרים;
- קובץ **tmpl1.cpp** מציג שימוש באלגוריתם הגנרי;
- קובץ **SafeArray.cpp** מכיל מחלקת תבנית המייצגת מערך בטוח, אשר אינו מאפשר גלישה מתחומו;
- קובץ **applic.h** מכיל את האפליקטורים לסוגיהם.

כדי להדגים **תכנות גנרי** (generic programming), נתחיל בפונקציה שאינה תבנית, אשר הופכת את סדר האיברים במערך נתון. עוד נראה כיצד להפוך פונקציה זו לפונקציית תבנית גנרית. אכן, זו שיטה טובה לפיתוח פונקציות תבנית גנריות. כאשר אתה מתקשה לכתוב פונקציה כזו, אפשר להתחיל בגירסה ראשונה שתפעל על מערך של מצביעים לשלמים, ולאחר מכן, כאשר האלגוריתם פועל לשביעות רצונך, להופכו לתבנית.

```
void reverse(int *begin , int *end)
{
    while (begin != end) {
       --end;
       if (begin == end)
         break;
       swap(*begin, *end);
       ++begin;
    }
}
```

הגירסה הראשונה של האלגוריתם פועלת על תחום מערך שלמים. התחום נתון על ידי מצביעים לשלמים המציינים את תחילתו וסופו של התחום. בגישה זו אנו יכולים להפוך את סדר האיברים בתחום כלשהו של המערך, ולאו דווקא בכל המערך. אפשר להפוך איברי תחום, שהוא רשימה, המוגדר על ידי שני איטרטורים.

האלגוריתם הופך את הערכים של תחילת התחום וסופו של התחום בלולאה, כשבכל שלב של הלולאה מזיזים את שני הקצוות של התחום אחד כלפי השני. במילים אחרות, המצביע להתחלת התחום מוגדל והמצביע לסוף התחום מוקטן. הלולאה מסתיימת כשהמצביע לתחילת התחום מגיע עד המצביע לסוף התחום. דבר חשוב באלגוריתם זה היא האפשרות להגדיל ולהקטין מצביעים. כלומר, האלגוריתם דורש שהמצביעים יהיו כאלה שעבורם קיימים האופרטור "++" והאופרטור "--", עובדה חשובה במיוחד עבור פונקציית תבנית.

לאחר שהבנו היטב את האלגוריתם לפונקציה רגילה, נוכל להמיר את הפונקציה **לפונקציית תבנית גנרית** (generic template function). הואיל ופונקציה גנרית אינה תלויה בסוג המצביע שהיא מקבלת, עלינו להפוך את סוג המצביע לפרמטר של התבנית. לכן, תחילת התחום וסופו הם מצביעים, או איטרטורים. נתייחס

לאיטרטורים כמצביעים, ואם נספק להם אופרטורים כגון "*" ואופרטורי הגדלה והקטנה ("++", "--"), יוכל האלגוריתם לפעול גם על אוספים. נראה זאת בהמשך.

כתוצאה, נקבל שהפונקציה reverse היא פונקציית תבנית שמקבלת סוג של איטרטור (מצביע מיוחד) והופכת את סדר האובייקטים במערך, בדיוק כמו הפונקציה הרגילה. לאחר המרה זו נקבל את פונקציית התבנית הבאה:

```
template <class Iter>
void reverse(Iter begin , Iter end)
{
   while (begin != end) {
      --end;
      if (begin == end)
      break;
      swap(*begin, *end);
      ++begin;
   }
}
```

הפונקציה reverse שהתקבלה כאן, יכולה כעת להפוך את סדר איברי מערך של אובייקטים, או של סוגים בסיסיים של C++, כגון שלמים או ממשיים. פונקציה זו יכולה גם לסרוק מחלקה, כגון רשימה, שמספקת איטרטור המאפשר לקדמו כמו מצביע, וגם מאפשר לבצע פעולת "*".

### 7.2.1.2.3 דוגמה - אלגוריתם מיון

נניח שמוטל עלינו לכתוב אלגוריתם למיון מערך. נבחר באלגוריתם **מיון מהיר** (quick sort), אשר נוקט בשיטת החלוקה וטיפול בתת-בעיות קטנות יותר. זוהי שיטת **'הפרד ומשול'**, ובלועזית: divide and conquer.

**האלגוריתם** מורכב מלולאה חיצונית ושתי לולאות פנימיות. הלולאה החיצונית מתבצעת כל עוד התחום (l, h) אינו ריק. בלולאה החיצונית בוחר האלגוריתם אובייקט במערך כאובייקט אמצעי (במקרה שלנו נבחר האובייקט הראשון). לאחר מכן, מתחיל האלגוריתם לסרוק את המערך מתחילתו לכיוון סופו, עד שהוא מוצא אובייקט הגדול מהאובייקט האמצעי (לולאה פנימית ראשונה). לאחר מכן, מתחיל האלגוריתם מסוף המערך כלפי תחילת המערך (לולאה פנימית שנייה). הלולאה ממשיכה עד שנמצא אובייקט הקטן מהאובייקט האמצעי. בסיום פעולתן של שתי הלולאות נמצאו שתי נקודות בתחום המכילות אובייקטים שמפרים את הסדר, יחסית לאובייקט האמצעי, ולכן יש להחליף ביניהם. פעולה זו מתבצעת עד שהסמנים (מצביעים או איטרטורים) של האיברים במערך מצטלבים.

לאחר כל זאת יש לנו שני תת-מערכים, שבאחד מהם נמצאים אובייקטים הקטנים מן האובייקטים שבאחר. זו הסיבה שאפשר להפעיל את האלגוריתם ברקורסיה על כל אחד מחלקי המערך, כדי לקבל את המערך הממוין באותה הצורה.

כדי להמחיש את האלגוריתם, נפַתֵח אותו תחילה עבור מערך של שלמים:

```
void quick_sort(int *begin, int *end)
{
    // median is first object
     int *l = begin, *h = end;
     if (begin < end) {
        while (1) {
            // find objects to swap
            while (*l < *begin && l<h) l++;
            while (*h > *begin && l<h) h--;
            if (l>=h) break;
            swap(*l, *h);
        }
        swap(*begin, *l);
        quick_sort(begin, l-1);
        quick_sort(l+1, end);
     }
}
```

לאחר שראינו מיון בשיטת גרסת המצביעים, נעבור לגירסת מיון בשיטת פונקציות התבנית. נמיר את המצביעים לאיטרטורים, כפי שעשינו עד כה. הפונקציות בדוגמה זו פועלות על תחום הנתון על ידי שני איטרטורים, התחלה וסוף. ההתייחסות לאיטרטורים כאן היא כמו למצביעים, ולכן אפשר להפעיל פונקציות אלו על אובייקטים בסיסיים של השפה, כגון שלמים, או על אוספים כגון מערכים.

הפונקציה quick_sort תהיה פונקציית התבנית ליישום אלגוריתם המיון. היא מקבלת איטרטור לתחילת המערך וסופו, ומבצעת את אלגוריתם המיון שתואר קודם לכן על התחום הנתון. גם פונקציה זו היא פונקציית תבנית.

הפונקציה quick_sort קוראת לפונקציה swap, כדי להחליף בין שני אובייקטים. swap מקבלת את סוג האובייקטים שיש להחליף כפרמטר של התבנית, אבל לפונקציה quick_sort אין סוג כזה. לאחר שראינו את האלגוריתם עבור תחום מסוים במערך של שלמים (כאשר התחום נתון על ידי שני מצביעים), אפשר להפוך את הפונקציה **לפונקציית תבנית גנרית.**

יש מצביעים לתחילת ולסוף התחום הנתון למיון. כשרוצים פונקציה גנרית, צריך להפוך את סוג המצביעים לפרמטר של התבנית. נקבל את הפונקציה הבאה:

```
template <class Iter>
void quick_sort(Iter begin, Iter end)
{
    // median is first object
     Iter l = begin, h = end;
     if (begin < end) {
        while (1) {
```

```
            // find objects to swap
            while (*l < *begin && l<h) l++;
            while (*h > *begin && l<h) h--;
            if (l>=h) break;
            swap(*l, *h);
        }
        swap(*begin, *l);
        quick_sort(begin, l-1);
        quick_sort(l+1, end);
    }
}
```

ההנחה שבהפעלת פונקציה זו היא שקיים אופרטור "*" שמחזיר אובייקט כשהוא מופעל על האיטרטור. דבר זה נגזר מפעולת הערך (*) על מצביע שהיה לנו בפונקציה הראשונה שהגדרנו (הגירסה הראשונה של האלגוריתם) שפעלה על המצביעים.

ההנחה שקיים אופרטור "*" מאפשרת שני דברים. האחד, חיסכון במספר הפרמטרים של פונקציית התבנית quick_sort. אחרת, היה עלינו לספק גם פרמטר המתאר את סוג האובייקט שהאיטרטור מצביע עליו. כאן מסיק המהדר את סוג האובייקט, ומייצר בעת הצורך את פונקציית התבנית (swap) שמתאימה לסוג האובייקט.

והדבר השני, פונקציית המיון יכולה לסרוק גם מערך של מצביעים וגם מחלקות עם איטרטורים, שיש בהם אופרטור "*" ואופרטור המאפשר סדר גדול (>) או קטן (<), בין אובייקטים שהאיטרטורים (או מצביעים) מתייחסים אליהם.

### 7.2.1.2.4 דוגמה - פונקציית הדפסה

הפונקציה print אף היא פונקציית תבנית שהפרמטר שלה הוא איטרטור. הפונקציה מקבלת איטרטורים לתחילת וסוף המערך בהתאמה, ומדפיסה את איברי המערך לפלט הסטנדרטי. הפונקציה מתחילה באיבר הראשון ומקדמת את האיטרטור עד שהיא מגיעה לסוף המערך.

האיטרטור end מצביע לאלמנט אחד **אחרי** סוף אוסף האובייקטים להדפסה. הפונקציה אינה מניחה שקיים אופטור ">" או "<" ולכן אפשר להשתמש בה לאיטרטורים הדומים לאלה של הרשימה מהפרקים הקודמים.

כמו בפונקציית המיון, כך גם פונקציית ההדפסה משתמשת באופרטור "*" כדי לקבל את תוכן האיטרטור. לכן, אפשר להשתמש בפונקציה זו כדי להדפיס מערכים, או כל אוסף שיש לו איטרטור, אם אופרטור "*" מתאים. גם זו פונקציה גנרית שיכולה לפעול על כל מערך או אוסף מתאים.

```
template <class Iter>
void print(Iter begin, Iter end)
{
    while (begin != end) {
      cout << *begin << " ";
```

```
        begin++;
    }
    cout << endl;
}
```

### 7.2.1.2.5 דוגמה - שימוש בפונקציות התבנית

הפונקציה הראשית (קובץ **tmpl.cpp**), מדגימה את השימוש בפונקציות אלו. מוגדר מערך של שלמים. המערך מודפס ולאחר מכן, ממוין ומודפס שנית. המצביע end מצביע לאלמנט אחד אחרי סוף המערך. לפיכך, כשמוסרים מצביע זה לפונקציית המיון, יש להפחית אחד. הממשק לשתי הפונקציות שונה, אך ניתן לתקן בעיה זו (תרגיל 1).

```
int main()
{
    int arr[] = { 1,5,3, 39, 8, -5, 25, 20, 4, 7,6, 9, 0};
    int *end = arr + sizeof(arr) / sizeof(int);
    print(arr, end);
    quick_sort(arr, end-1);
    print(arr, end);
    reverse(arr, end);
    print(arr, end);
    return 0;
}
```

פונקציית המיון שראינו, משתמשת באופרטור "<" ולכן לא נקבל את המיון הלקסיקוגרפי הנדרש עבור מצביעים למחרוזות. לצורך כך יש להעביר לפונקציה **פונקציית השוואה** להשוואה בין האובייקטים השונים. נראה דוגמה בהמשך הפרק.

מומלץ לקורא לבחון את ההבדל בממשק של הפונקציה quick_sort לפונקציית המערכת qsort של C. נוסיף לפונקציה שיפורים בהמשך, כשנלמד על **מחלקות תבנית** (template classes).

הפונקציה quick_sort יעילה מבחינת זמן הריצה לפחות כמו פונקציית הספרייה qsort, משום שאינה משתמשת במנגנון הפולימורפי ב-++C. לעומת זאת, לכל סוג של משתנה מייצר המהדר פונקציה כזו מחדש, דבר הגורם ליצירה מרובה של קוד. עם זאת, השימוש בפונקציה זו נעשה בצורה טבעית, ללא צורך לבצע המרות מסוג לסוג.

## 7.2.2 מחלקות תבנית

מחלקות תבנית מכילות פרמטר עבור סוג האובייקטים שהן מטפלות בהם. למשל, המחלקה list שפיתחנו בפרק קודם, אינה תלויה בסוג האובייקטים ברשימה. לכן, אפשר לכתוב את המחלקה באופן גנרי, כך שנקבל רשימה עבור כל סוג של אובייקט או משתנה (למשל שלמים או תווים), ללא כל צורך לבצע המרות של מצביעים, וללא צורך לרשת מהצומת הבסיסי של הרשימה.

מחלקות המכילות אובייקטים אחרים ומבצעות פעולות של הכנסת אובייקטים או ביטולם, ומעבר על אוסף האובייקטים במחלקה, נקראות **מכולות** (containers), **אוסָפים**, או **מיכלים**. מחלקות אלו מאופיינות בכך שיש להן פונקציונליות שאינה תלויה בסוג האובייקטים בהן הן מטפלות. על כן הן מועמדות טבעיות להפוך למחלקת תבנית.

הצורה להגדרת מחלקת תבנית, דומה לצורה בה מגדירים פונקציית תבנית. למשל:

```
template <class Type>
class X {
...
};
```

כמו במקרה של פונקציית תבנית, גם כאן אנו משתמשים במילת המפתח **template** ולאחריה רשימת פרמטרים של התבנית המופיעים בין הסימנים "< >". מספר הפרמטרים שאפשר להציב ברשימה אינו מוגבל. כל משתנה מופיע בתוספת המילה השמורה **class**. כשיש מחלקת תבנית כזו, חייב להיות לכל אחד מהפרמטרים של התבנית שימוש במחלקה לפחות פעם אחת.

בסעיף הקודם הגדרנו פונקציית מיון שהסתמכה על קיום אופרטור "<" שבדק את הסדר בין שני אובייקטים. כעת ניתן להגדיר מחלקת תבנית המבצעת תפקיד זה. המחלקה less יכולה לעשות זאת.

```
template <class T>
class less {
public:
      int check(T v1, T v2) const
      { return (v1 < v2); }
};
```

למחלקה less אין כל שדות בחלק הפרטי, והיא משתמשת באופרטור "<" לכל שני אובייקטים מסוג T, שהוא פרמטר התבנית. אפשר להגדיר מחלקה זו כ-struct, כך:

```
template <class T>
struct less {
      int check(T v1, T v2) const
      { return (v1 < v2); }
};
```

קיבלנו מחלקת תבנית הבודקת את הסדר של שני אובייקטים כלשהם. הפונקציה check מחזירה ערך אמת (true) אם האובייקט הראשון קטן מהשני. כעת אנו יכולים להשתמש במחלקה זו באופן הבא:

```
int x1, x2;
less<int> li;   // line 2
x1 = 1;
x2 = 2;
if (li.check(x1, x2))
```

```
      cout << "x1 < x2" << endl;
else
      cout << "Someting very strange happened" << endl;
```

השורה השנייה מגדירה אובייקט של המחלקה less. כשמוגדר אובייקט של מחלקת תבנית, יש לספק את הפרמטרים של התבנית למחלקה. הפרמטרים האלה מסופקים לאובייקט בתוך ה-"<>", שזו רשימת הפרמטרים. דבר זה קורה בשורה השנייה של קטע הקוד שלמעלה, שבה מסופק למחלקת התבנית הסוג int. השורה השנייה גורמת למהדר לייצר את הקוד המתאים לאובייקט li. ההצהרה על מחלקת התבנית אינה יוצרת קוד, אך **השימוש** בה יוצר קוד: הגדרת אובייקט השייך למחלקת התבנית. הגדרת אובייקט נקראת בלועזית **instantiation**.

## 7.2.2.1 אובייקטי פונקציות

**אובייקטי פונקציות** (function objects) מתנהגים כפונקציות לכל דבר. הם אינם חייבים להיות מחלקות תבנית. יחד עם זאת, יש יתרון לאובייקטי פונקציות השייכים למחלקת תבנית.

למשל, נגדיר אופרטור "()" למחלקה less, ונקבל מחלקה שהאובייקטים שלה מתנהגים כמו פונקציות. היתרון הוא בכך שנוכל לספק אובייקט מסוג זה לפונקציות תבנית שמצפות לקבל מצביע לפונקציה, או פונקציה שבודקת את הסדר בין אובייקטים.

```
template <class T>
struct less {
      int operator()(T v1, T v2) const
      { return (v1 < v2); }
};
```

עתה ייראה כך הקוד הבוחן את היחס בין שני אובייקטים שונים:

```
int x1, x2;
less<int> li;
x1 = 1;
x2 = 2;
if (li(x1, x2))
      cout << "x1 < x2" << endl;
else
      cout << "Someting very strange happened" << endl;
```

li, האובייקט של מחלקת התבנית, מתנהג בדיוק כמו פונקציה. כשמספקים לו שני פרמטרים, מפעיל המהדר את אופרטור הקריאה לפונקציה, אשר שייך לאובייקט הנוכחי. היתרון של אובייקט המתנהג כפונקציה הוא בכך שאפשר להעביר אותו לפונקציה שמצפה לקבל פונקציה אחרת.

לדוגמה, אפשר לשנות את הגדרת פונקציית המיון כך שזו תקבל פונקציה, או אובייקט המתנהג כפונקציה, כדי לבדוק אם האובייקט הראשון קטן מהשני. נוסיף לפונקציית המיון פרמטר תבנית נוסף, המייצג את פונקציית ההשוואה בין האובייקטים. כעת אפשר להעביר לפונקציה זו אובייקט המתנהג כפונקציה, או פונקציית השוואה.

```
template <class Iter, class Comp>
void quick_sort1(Iter begin, Iter end, Comp cmp)
{
   // median is first object
    Iter l = begin, h = end;
    if (begin < end) {
       while (1) {
           while (cmp(*l, *begin) && l<h) l++;
           while (cmp(*begin, *h) && l<h) h--;
           if (l>=h) break;
           swap(*l, *h);
       }
       swap(*begin, *l);
       quick_sort1(begin, l-1, cmp);
       quick_sort1(l+1, end, cmp);
    }
}
```

כפי שראינו קודם, להגדרת האובייקט less דמוי הפונקציה השתמשנו באופרטור "<", ולכן אובייקט זה אינו מתאים עבור מצביעים לתווים (מחרוזות). כעת אפשר להעביר את הפונקציה strcmp לפונקציית המיון. דרך שנייה היא להגדיר מחלקה נוספת.

## 7.2.2.2 שימוש בפרמטר קבוע למחלקת תבנית

ראינו כבר שיש בעיה כשמשתמשים במחלקה less עבור מחרוזות, כי השימוש באופרטור "<" אינו תקף עבור מצביעים לתווים. C++ מאפשרת להגדיר מופע של מחלקת תבנית. בצורה זו אפשר להגדיר מופע של המחלקה less המתאים למצביעים לתווים:

```
struct less<char *> {
      int operator()(char *s1, char *s2) const
      { return (strcmp(s1, s2) < 0); }
};
```

כשמגדירים מופע מיוחד של מחלקת תבנית, אין המהדר יוצר את הקוד למופע זה אם הוא מגלה שנעשה כבר שימוש בפרמטר התבנית הנתון. כאשר המהדר מגלה שימוש במחלקת התבנית עבור *char למשל, הוא ישתמש במה שהגדרנו, ולא ייצר מחלקה נוספת עבור מצביע לתווים.

עם ההגדרות שלנו אנו יכולים כעת להשתמש בפונקציית המיון בדרך הבאה:

```
int main()
{
    int arr[] = { 1,5,3, 39, 8, -5, 25, 20, 4, 7,6, 9, 0};
    int *end = arr + sizeof(arr) / sizeof(int);
    char *sarr[] = {  "shimon", "leora", "nirritt",
       "ornit", "yaffa", "riveka", "israel" };
    char **send = sarr + sizeof(sarr) / sizeof(char*);
    print(arr, end);
    quick_sort1(arr, end-1, less<int>());
    print(arr, end);

    print(sarr, send);
    quick_sort1(sarr, send-1, less<char*>());
    print(sarr, send);
    return 0;
}
```

הגדרנו כאן שני מערכים. המערך הראשון מכיל שלמים והמערך השני הוא של מצביעים לתווים. כעת, כשקוראים לפונקציית המיון יש לספק אובייקט המבצע את ההשוואה. במקרה של מערך השלמים, מסופק אובייקט המבצע השוואה בין שלמים.

הביטוי less<int>() מספק אובייקט כזה, ולעומתו הביטוי less<int> הוא **סוג** (type) של אובייקט (כמו ש-int הוא סוג של משתנה). כאשר יש למיין את מערך המצביעים לתווים, מוסרים לפונקציית המיון את אובייקט ההשוואה less<char*>(). במקרה זה המהדר אינו יוצר קוד למחלקה זו, משום שהגדרנו קוד זה באופן מפורש.

## 7.2.2.3 דוגמה - רשימה מקושרת

בפרקים הקודמים הגדרנו **רשימה מקושרת** (linked list). בסעיף זה נראה כיצד ניתן להשתמש בתבניות ברשימה המקושרת, כדי לתמוך בכל סוג של אובייקט ברשימה. כמו כן, נראה שפונקציות כגון print תפעלנה גם עבור הרשימה המקושרת.

### 7.2.2.3.1 רשימת תבנית - ניסיון ראשון

כדי להגדיר רשימה כפולה בכל צומת ברשימה, דרושים לנו מצביעים לתחילת וסוף הרשימה, וערך שיש לשמור בצומת. לפיכך, צומת ברשימה כזו ייראה כך:

```
template <class T>
struct list_node {
      list_node<T> *next;
      list_node<T> *prev;
      T             value;
};
```

ההגדרה של מצביעים לצומת הקודם ולצומת הנוכחי נעשית באמצעות פרמטר התבנית T. מכיון שבהגדרת המחלקה אין אנו יודעים את סוג פרמטר התבנית, עלינו להשתמש בשם שלו בתוך "<>".

הרשימה עצמה מכילה מספר לא קבוע של צמתים כאלה, ומוגדרת כך:

```
template <class T>
class list {
	list_node<T> *root;
	...
public:
	...
	void append(const T &val) {
		list_node<T> *p = new list_node<T>;
		p->value = val;
		...
	}
};
```

בכל פעם שנשתמש במחלקה list עבור סוגים שונים, ייצר המהדר את הקוד מחדש עבור כל סוג. לכן ייצר קטע הקוד הבא ארבע מחלקות שונות, שהקוד שלהן משוכפל.

```
list<int> li;
list<double> ld;
list<float> lf;
list<foo> lfoo;
```

אפילו התוכנית הקטנה ביותר תהיה בעלת נפח גדול. מכאן ברור, שגישה זו אינה יעילה ועלינו לנקוט בדרך פעולה שונה.

### 7.2.2.3.2 שימוש בירושה ממחלקה שאינה תבנית

ראינו שמנגנון התבנית מייצר מחלקה ואת כל הקוד שלה בכל שימוש בתבנית עם סוג שלא השתמשו בו עדיין בתוכנית הנוכחית. תכונה זו גורמת לייצור קוד רב. הדרך הטובה להתגבר על בעיה זו היא להגדיר מחלקה בסיסית שמבצעת את השירותים הבסיסיים של הרשימה. כשיש לנו מחלקה בסיסית כזו, נגזור ממנה מחלקה ליישום השירותים המסוימים של הרשימה, על פי סוג התבנית.

כאשר גוזרים **מחלקת תבנית** (template class) ממחלקה אחרת (שאף היא יכולה להיות מחלקת תבנית), מתקיימים כל חוקי הגזירה לגבי המחלקות המעורבות ביחס הירושה (או הגזירה).

בהתאם לכך, תהיינה לנו **מחלקות בסיסיות** (base classes) שיספקו את שירותי הרשימה והמעבר על הרשימה. מחלקות התבנית ירשו מהמחלקות האחרונות וישתמשו בשירותים אלה. לכן, משלבים ירושה ותבניות כדי לקבל מחלקה חדשה.

בצורה זו אנו מטפלים בבעיית ייצור הקוד על ידי המהדר, אשר אינו צריך עוד לייצר קוד עבור המחלקות הבסיסיות שאינן מחלקות תבנית.

בפרקים הקודמים כבר הגדרנו רשימה. נשתמש ברשימה שיצרנו בפרקים הקודמים אך נשנה את השמות בצורה הבאה:

- **list** - ייקרא מעתה base_list.
- **list_iter** - ייקרא מעתה base_list_iter.
- **link** - ייקרא מעתה base_link.

שינוי השמות נעשה רק למען נוחות המשתמש במחלקה זו ואינו מחויב. בספריות שונות נמצא ששם מחלקת תבנית היורשת ממחלקה שאינה כזו, הוא **tlist**, כלומר תוספת של t לפני השם.

המחלקות הקודמות מומרות בצורה הבאה לשמות החדשים שלהן (המחלקות נמצאות בקבצים **list.h**, **list.cpp**):

```
class base_link {
// as before
};

class base_list_iter {
// as before
public:
// as before
      base_link *get_current()
      { return current; }
// as before
};
```

למחלקה base_list_iter הוספנו פונקציה אחת בלבד, המאפשרת לקבל את הצומת הנוכחי. פונקציה זו התוספה רק מטעמי נוחות (במקום להפעיל את האופרטור "*").

```
class base_list {
// as before
public:
// as before
      int size() const { return nobjects; }
};
```

למחלקה base_list הוספנו פונקציה שמחזירה את מספר האלמנטים שנמצאים ברשימה. כל שאר הפונקציות נשארו כשהיו.

נירש את המחלקה החדשה מהמחלקה של הרשימה שהיתה לנו בפרקים הקודמים. נעשה זאת כך:

```
template <class T>
class list : public base_list {
```

```
    friend class list_iter;
    struct list_node : public base_link {
      T val;
      list_node(const T &v) : val(v) {}
    };
public:
    typedef list<T>::list_node node_type;

    list()               // empty constructor
    {}
    ~list() { delete_all(); }
    void delete_all()   // deletes all nodes on the list
    { while (size() > 0)
      delete (node_type*)base_list::remove_head(); }
    void append(const T &val)
    { base_list::append(new node_type(val)); }
    void insert(const T &val)
    { base_list::insert(new node_type(val)); }
    void insert(base_list_iter &pos, const T &val)
    { base_list::insert(*pos, new node_type(val)); }
    T head()             // gets the head
    { node_type *hd = (node_type*)base_list::head();
      return hd->val; }
    T tail()             // gets the tail
    { node_type *tl = (node_type*)base_list::tail();
      return tl->val; }
    T remove_head() {
       node_type *tl = (node_type*)base_list::remove_head();
       T rval(tl->val);
       delete tl;
       return rval;
    }
    T remove_tail() {
       node_type *tl =  (node_type*)base_list::remove_tail();
       T rval(tl->val);
       delete tl;
       return rval;
    }
    void remove(const T &val);//remove all occernces of val
    //
    // list iterator
    //
    class list_iter : public base_list_iter {
    public:
```

```
        list_iter(base_link *bl) : base_list_iter(bl)
        {}
        list<T>::list_iter &operator++()
        { next(); return *this; }
        list<T>::list_iter operator++(int) {
             list<T>::list_iter self(*this);
             next();
             return self;
       }
       list<T>::list_iter &operator--()
       { prev(); return *this; }
       list<T>::list_iter operator--(int) {
             list<T>::list_iter self(*this);
             prev();
             return self;
       }
       T operator*() {
             node_type *lnd = (node_type*)get_current();
             return lnd->val;
       }
    };
    typedef list<T>::list_iter iter_type;

    iter_type begin() { return
iter_type(base_list::head()); }
    iter_type end()   { return
iter_type((base_link*)nil()); }
};
```

### מחלקות מקוננות בפנים הרשימה

בתוך מחלקת הרשימה (list) מוגדרות שתי **מחלקות מקוננות** (nested classes): list_node ו-list_iter. כשיש מחלקות מקוננות, אין גישה אליהן פרט למחלקה המכילה אותן. גם למחלקה המכילה אין גישה מיוחדת לשדות של המחלקות המקוננות, או המוכלות.

המחלקה list_node מייצגת את הצומת ברשימה. המחלקה list_iter מייצגת את סורק הרשימה.

המחלקה list_iter מוגדרת בחלק הציבורי של המחלקה list, לכן יש גישה לאיטרטור לפונקציות, או אובייקטים מחוץ לרשימה. לעומת זאת, אם מחלקה מקוננת מוגדרת בחלק הפרטי (כמו list_node) של מחלקה אחרת, אין לאף פונקציה, או מחלקה אחרת, גישה למחלקה זו פרט לפונקציות של המחלקה המכילה, או של חברים של המחלקה המכילה.

מחלקות מקוננות המוגדרות בחלק השמור של המחלקה המכילה, נועדו לאפשר למחלקות היורשות מהמחלקה המכילה גישה אל השדות הציבוריים שלהן בלבד.

מחלקות מקוננות שמוגדרות בחלק הציבורי של המחלקה המקוננת, נועדו לאפשר גישה לכל פונקציה ולכל מחלקה אחרת, לשדות הציבוריים שלהן בלבד.

בכל מקרה, מוגבלת הגישה רק לחלקים הציבוריים של המחלקה המקוננת, או המכילה. כדי לאפשר רמת גישה גבוהה יותר יש להשתמש במנגנון החבר, דבר שאינו מומלץ, פרט למקרים מיוחדים (כמו, איטרטורים).

היתרון של הגדרת מחלקה מקוננת בכך שאנו לא מזהמים את מרחב השמות הגלובלי של התוכנית. דבר זה חשוב מאוד בתוכניות גדולות המשתמשות במספר גדול של ספריות, עם סבירות גבוהה להתנגשויות בין שמות של ספריות. ב-++C יש כלי אחר המטפל במקרה זה והוא **namespace**. נלמד אודות כלי זה בפרקים הבאים. בכל מקרה, יש לדעת שבעת כתיבת שורות אלו אין מהדרים התומכים באפשרות זו.

### צומת ברשימה

**צומת ברשימה** (node) הוא מחלקת תבנית המכילה אובייקט מסוג T שהוא הפרמטר של התבנית. לפיכך, הצומת ברשימה תלוי בסוג האובייקט שיהיה ברשימה. לכן, הצומת ברשימה משתמש בפרמטר של התבנית T, כדי לייצג את הערך שנשמר בצומת הנוכחי.

הואיל והצומת ברשימה מצוי בחלק הפרטי שלה, הגדרנו מחלקה זו **כרשומה** (struct). שדות המחלקה הזו פתוחים בפני כל הפונקציות של המחלקה רשימה. הבנאי של מחלקה זו מאתחל את האובייקט בצומת (val) בעזרת בנאי ההעתקה של האובייקט בצומת. לכל אובייקט ב-++C מובטח שיהיה בנאי כזה, ולכן הגדרנו את הבנאי של הצומת בצורה זו.

הגדרה אחרת לבנאי יכולה להיות:

```
list_node(const T &v) { val =v; }
```

הבעיה בהגדרה זו היא שאנו מניחים שקיים **בנאי ברירת מחדל** (default constructor) לאובייקט שבצומת, וכמובן שלא מובטח לנו קיומו של בנאי כזה. בעיה אחרת היא, שאפילו אם בנאי זה אמנם קיים, תתבצע תחילה קריאה לבנאי ברירת המחדל, ורק לאחר מכן קריאה לאופרטור ההשמה של האובייקט בגוף הבנאי של הצומת ברשימה. דבר זה אינו יעיל, כי הוא מחייב קריאה נוספת לפונקציה (בנאי ברירת המחדל).

### האיטרטור

גם האיטרטור list_iter הוא מחלקה מקוננת, אשר יורשת מהאיטרטור הבסיסי. גם היא מחלקת תבנית המשתמשת בפרמטר T של מחלקת התבנית המכילה אותה, list.

כמו באיטרטור הבסיסי, גם כאן מוגדרים האופרטורים לקידום האיטרטור, או חזרה לאובייקט קודם. שני סוגי פעולות אלה מוגדרים בשתי הסמנטיקות של פעולות **prefix** ו-**suffix**. בסוג הראשון מקדמים **תחילה** את האיטרטור, ולאחר מכן מחזירים ייחוס אליו. בסוג השני מחזירים ייחוס לאיטרטור, **לפני** פעולת הקידום.

כזכור, פועל האיטרטור כמצביע, כדי לאפשר לאלגוריתמים גנרים לפעול על רשימות. לפיכך, קיים אופרטור "*" שמחזיר את תוכן הצומת הנוכחי באיטרציה, כשהסוג המוחזר הוא מסוג פרמטר התבנית.

**רשימה**

הרשימה עצמה יורשת מהרשימה הבסיסית. אבל כשרוצים להכניס ערך כלשהו לרשימה, מוקצה צומת חדש שמחזיק את הערך הנדרש, והוא זה שמוכנס לרשימה. הטיפול בזיכרון מוטל עתה על הרשימה, ולא על המשתמש בה. לכן, הקוד של המשתמש ברשימה פשוט הרבה יותר.

כאשר הרשימה יוצאת מתחום ההגדרה, נקרא המפרק שלה ואז היא משחררת את כל הזיכרון שהקצתה במהלך חייה. שחרור הזיכרון מתרחש גם כשמבטלים צומת כלשהו ברשימה. פונקציות המבטלות את האיבר הראשון, או האחרון, צריכות לשחרר את הזיכרון שאיחסן את הצומת הזה.

השינוי הגדול הוא כאשר רוצים לבטל אובייקט ברשימה. הערך שרוצים לבטל נמסר לפונקציה remove. היא מגדירה איטרטור, סורקת את הרשימה ומבטלת את האובייקטים הזהים לאובייקט הנתון לה.

מכיון שהמפרק של הצומת הבסיסי base_link אינו וירטואלי, כשמפעילים את אופרטור delete יש להמיר את הכתובת לכתובת המתאימה לצומת ברשימה שיורש מהצומת הבסיסי. פעולה זו נעשית תמיד כשמבטלים אובייקט מהרשימה.

```
template <class T>
void list<T>::remove(const T &key)
{
    iter_type ls(begin()), le(end());
    while (ls != le) {
        if (*ls == key) {
            // found remove from the list
            base_link *blp = ls.get_current();
            ++ls;
            base_list::remove(blp);
            delete (node_type*)blp;
        }
        else
            ++ls;
    }
}
```

כאשר מגדירים פונקציית מחלקת תבנית מחוץ למחלקה עצמה, יש להגדיר את הפונקציה בתוספת הפרמטרים של התבנית. למשל, עבור רשימה היה לנו פרמטר אחד של תבנית T, ולכן עלינו להגדיר את הפרמטר הזה גם בהגדרת הפונקציה. בנוסף לזה, שם הרשימה הוא list<T>, כאשר T הוא הפרמטר של התבנית. למשל, אם היו שני פרמטרים, היינו צריכים לכתוב: list<T1, T2>.

ברשימה יש שתי הגדרות סוגים המתבצעות בעזרת **typedef**. דבר זה נעשה כדי לחסוך בכל פעם את הכתיבה מחדש של סוג הצומת ברשימה, או של האיטרטור, כי הגדרה מורכבת כזו עלולה לגרום לשגיאות. אם נרצה לשנות את סוג הצומת, או האיטרטור, נשנה רק את typedef.

כמות הקוד שהיינו צריכים לכתוב, לו היינו כותבים את הרשימה ואת האיטרטורים מהתחלה כתבניות, היתה גדולה מאוד, ולכן גם הקוד שהיה המהדר יוצר היה גדול. אנו חוסכים כאן קוד רב. מומלץ להשתמש במידת האפשר במחלקה בסיסית שאינה תבנית, ולעשות במחלקה הבסיסית את רוב העבודה. המחלקות היורשות צריכות להיות קצרות, וכך לחסוך קוד המיוצר עבור כל שימוש במחלקה. נוסף לכך, יצירת קוד גורמת לזמן הידור ארוך יותר, כי המהדר צריך ליצור את הקוד.

בנוסף, לא הגדרנו פונקציות וירטואליות ולכן הטיפול ברשימה יעיל ומהיר מאוד. אכן, בעזרת תבניות בצורה מבוקרת אפשר להגיע לתוכנה יעילה ובאיכות גבוהה מאוד. כשאנו מנצלים את אפשרות לרשת את מחלקת התבנית ממחלקה בסיסית שאינה מחלקת תבנית, אנו מונעים ייצור קוד מיותר על ידי המהדר.

### פונקציית מיון לרשימה

הפונקציה הגנרית **quick_sort** אינה יכולה לפעול על רשימה, מכיון שהיא משתמשת באופרטורים שדורשים סדר בין מצביעים. נשתמש ב**אלגוריתם גנרי** המאפשר למיין את הרשימה בשיטת **insertion_sort**.

האלגוריתם עצמו פועל בשתי לולאות, חיצונית ופנימית. ראינו שהאלגוריתם מקבל תחום למיון ופונקציה או אובייקט-פונקציה, לצורך השוואות. בכל שלב של הלולאה החיצונית מוקטן התחום הכללי באובייקט אחד. דבר זה נעשה על ידי קידום התחלת התחום, האיטרטור start. בכל שלב של האלגוריתם מועבר האובייקט הנוסף (המסומן על ידי start) למקומו בתחום.

הלולאה הפנימית מתחילה עם האובייקט הראשון והשני, ובכל שלב כוללת תחום גדול יותר. בשלב השני, של הלולאה הפנימית, מטופלים שלושת האובייקטים הראשונים. בשלב השלישי מטופלים ארבעת האובייקטים הראשונים. בכל שלב של הלולאה הפנימית, התחום הקודם כבר ממוין, לכן מכניסים את האובייקט הנוסף למקום המתאים לו.

```
template <class Iter, class Great, class Item>
void insertion_sort(Iter begin, Iter end,
                        Great grt, Item item)
{
    Iter start(begin);
    ++start;
    while (start != end) {
           Iter next(start), current(start);
           item = *start;
           --next;
```

```
            while (grt(*next, item)) {
                *current = *next;
                --current;
                if (current == begin) break;
                --next;
            }
            *current = item;
            ++start;
        }
}
```

אלגוריתם מיון זה הוא ברמת סיבוכיות O(n*n) (האם תוכל להוכיח זאת?), לכן הוא פחות יעיל מהאלגוריתם quick_sort. אלגוריתם זה אינו מחייב השוואה בין כתובות, לכן הוא מתאים לרשימה. כמו quick_sort גם אלגוריתם זה פועל על אזור הזיכרון הנתון לו על ידי התחום, ואינו מעתיק את הנתונים לאזור זיכרון חדש. כמו האלגוריתם הקודם, כך גם אלגוריתם זה יכול למיין מערכים בסיסיים, או מכולות.

### 7.2.2.3.3 שימוש ברשימה

כשנתונה רשימה, אפשר להשתמש בה בשני אופנים. בצורה הפשוטה נכלול בה ערכים כלשהם, שאנו מעתיקים אותם לצמתים שונים ברשימה.

השימוש השני הוא לכלול בה מצביעים, ולא ערכים. כשיש לנו רשימת מצביעים, ייתכנו מצביעים המשותפים למספר רשימות. במקרה זה, עלינו לנהל את הזיכרון בעצמנו, משום שהרשימה אינה מטפלת בזיכרון שהוקצה על ידי המשתמש בה. במקרה זה, קיים יתרון שאנו יכולים עדיין להשתמש ברשימה בצורה פולימורפית (למרות שהרשימה חסרת פונקציות וירטואליות).

#### שימוש ברשימה על ידי העתקה

כאשר משתמשים ברשימה על ידי העתקה, מגלים שכל ערך המוכנס בה מועתק אליה וברשימה יש רק העתק של הערך. לכן, היא אחראית להקצאה ושחרור של הזיכרון לכל צומת כזה. כמו כן, אין ערכים המשותפים למספר רשימות כאלו. כאשר הרשימה מחזיקה סוג מסוים ויחיד של אובייקטים, היא נקראת **רשימה הומוגנית** (homogeneous list). הרשימה שהגדרנו בצירוף האיטרטור יכולה להיות מועברת לאלגוריתמים שונים כמו print, מכיון שאיטרטור הרשימה מתנהג כמו מצביע רגיל של ++C. אפשר גם להפעיל את האלגוריתמים הגנרים האלה על כל מערך ב-++C. לדוגמה, הקוד הבא הוא חוקי:

```
int arr[] = { 1, 2, 5, 7, 3, 4, 6}, tmp;
int *arrend = arr + sizeof(arr)/sizeof(int);
list<int> li;
li.append(1);
li.append(2);
li.append(5);
```

```
li.append(7);
li.insert(3);
li.insert(4);
li.insert(6);
cout << "\n before insertion sort" << endl;
print(li.begin(), li.end());
insertion_sort(li.begin(), li.end(), great<int>(), tmp);
cout << "\n after insertion sort" << endl;
print(li.begin(), li.end());
cout << "\n array before insertion sort" << endl;
print(arr, arrend);
insertion_sort(arr, arrend, great<int>(), tmp);
print(arr, arrend);
```

למעשה, במערך arr יש סדרה של ערכים הזהה לערכים ברשימה. תחילה מודפסת הרשימה המקורית. אחר כך מפעילים עליה אלגוריתם של מיון, ולבסוף היא מודפסת שנית לפלט הסטנדרטי. תהליך דומה עובר על המערך. שים לב לכך שממיינים הן את המערך והן את הרשימה על ידי אותה פונקציית תבנית!

### שימוש ברשימת מצביעים

שימוש ברשימת מצביעים קשור בדרך כלל לשימוש פולימורפי ברשימה. בפרק הקודם בו למדנו על **פולימורפיזם** (polymorphism), ראינו דוגמה המציירת צורות על המסך. בדוגמה ההיא היתה רשימה של צורות מסוג שונה, כלומר, היו אובייקטים מסוג שונה. רשימה כזו נקראת **רשימה הטרוגנית** (heterogeneous list). כדי לצייר את הצורות, סרקנו את הרשימה והפעלנו את הפונקציה draw לכל צורה בנפרד. מכיון שהפעלנו את הפונקציה הזו דרך מצביע נחשבה הפעלה זו פולימורפית. כלומר, הפונקציה שהופעלה על ידי המערכת הפולימורפית היא זו שהתאימה את הפונקציה לסוג האובייקט.

אפשר להשתמש ברשימה שלנו כרשימה של מצביעים, אך אז עלינו לשחרר אותם בעצמנו בעת הצורך. נראה כיצד אותה הדוגמה, העוסקת בצורות וציורן, נראית בעזרת רשימת התבנית. נציג רק את קטעי הקוד המתאימים ואת אלה שישתנו בעת כתיבה בדרך זו. שאר הקוד זהה לדוגמה הקודמת. הקוד לתוכנית הנוכחית נמצא בקובץ **polym3.cpp.**

המחלקה Shape אינה יורשת מהצומת הבסיסי של הרשימה. רשימת התבנית מוגדרת כרשימת מצביעים ל-Shape, ולכן אין צורך בירושה הקודמת.

```
class Shape  {
public:
     Shape() {}
     virtual void draw(Screen &) {}
};
```

כמו בדוגמה הקודמת, גם כאן יורשות המחלקות המייצגות צורות שונות מהמחלקה Shape, שיש לציירן על המסך. מחלקות אלו מגדירות מחדש את פונקציית הציור.

```
class Point : public Shape {
      int x,y;
public:
      Point(int xo, int yo)  { x= xo; y=yo; }
      void draw(Screen &scr)
      { scr.put(x,y); }
};

class Line : public Shape {
      int x1, y1;
      int x2, y2;
public:
      Line(int xo1, int yo1, int xo2, int yo2)
      { x1 = xo1; y1 = yo1; x2 = xo2; y2 = yo2; }
      void draw(Screen &scr);
};
```

האלגוריתם של פונקציית הציור של המחלקה Line לא השתנה, ולכן אינו מפורט כאן.

האלגוריתם המצייר את כל הצורות על המסך מקבל רשימת צורות אפשריות. הפונקציה מגדירה איטרטור שמאותחל לתחילת הרשימה ומשתמשת בו כדי לסרוק את הצורות ברשימה כדי לצייר אותן זו אחר זו. כשמשתמשים ברשימת הצורות בדרך זו, אין כל צורך להמיר כתובות.

```
void draw(list<Shape*> &slist, Screen &scr)
{
      list<Shape*>::list_iter i = slist.begin();
      scr.clear();
      while (i != slist.end()) {
            Shape *sp = *i;
            sp->draw(scr);
            ++i;
      }
      scr.draw();
}
```

הפונקציה הראשית מגדירה את רשימת הצורות, יוצרת אותן, ואחר כך מכניסה אותן לרשימה. הצורות שנוצרו יכולות להופיע בשתי רשימות, או יותר. בדרך הזו של שימוש ברשימה אנו רואים כי המשתמש ברשימה היקצה את הזיכרון עבור הצורות שברשימה, ולכן מוטל עליו לשחרר את הזיכרון שנתפס. דבר זה אינו נעשה בקטע הקוד, ואני משאיר לקורא לעשות זאת כתרגיל. הרשימה רק מבטלת את הצמתים שהיא הקצתה.

```
int main()
{
      list<Shape*> shapes;
```

```
        Screen scr;
        shapes.append(new Line(10,1, 19, 11));
        shapes.append(new Line(40, 20, 55, 10));
        shapes.append(new Point(10, 20));
        shapes.append(new Point(15, 10));
        draw(shapes, scr);
        return 0;
}
```

מכיון שהרשימה אינה משחררת את המצביעים שהוכנסו לצמתים שלה, היינו רוצים לאפשר פונקציונליות זו בצורה מרוכזת, כדי שהדבר לא יידרש מכל משתמש ברשימה, שייאלץ לפתוח אותה מחדש. השאלה היא האם לפתח פונקציה שמשחררת מצביעים ברשימה, או האם ניתן לפתח מנגנון המאפשר להפעיל פונקציות על הרשימה?

## 7.2.3 אפליקטורים

ראינו שיש בעיה של שחרור זיכרון כשמשתמשים ברשימת מצביעים. **אפליקטורים** (aplicators) הם אובייקטים המאפשרים להפעיל פונקציה כלשהי על האובייקטים שנמצאים ברשימה. לכן, **אפליקטור** הוא פתרון כללי לבעיית השחרור. במידה שקיים אפליקטור כזה, אפשר להפעיל על הרשימה פונקציה שמשחררת אובייקט (ולמעשה, מפעילה אופרטור delete).

האפליקטור שנגדיר כאן הוא מחלקת תבנית הפועלת בתחום מסוים, כמו כל האלגוריתמים הגנרים שהגדרנו. האפליקטור נראה כך :

```
template <class Iter, class Func>
class applicator {
        Func func;          // the function to perform
public:
        applicator(const Func &fct) : func(fct)
        {}
        void apply(Iter begin, Iter end);
        int operator()(Iter begin, Iter end)
        { apply(begin, end); return 1;}
};
```

האפליקטור הוא **מחלקת תבנית** שמקבלת שני פרמטרי תבנית. הראשון, איטרטור שמתנהג כמצביע והשני, סוג הפונקציה שיש להפעיל. לאפליקטור יש שדה יחיד שהוא הפונקציה שתופעל על תחום נתון. הפונקציה יכולה להיות פונקציה אמיתית, או אובייקט המתפקד כפונקציה.

הפונקציה apply עוברת על תחום נתון ומבצעת את הפונקציה הנתונה לה, כל עוד הפונקציה הנתונה מחזירה ערך השונה מאפס. היתרון של הגדרה כזו לפונקציה apply הוא שאפשר להפעיל את האפליקטור על תחום נתון, ולא על כל המערך, או כל הרשימה. בכל הפעלה של האפליקטור נבחר תחום חדש.

```
template <class Iter, class Func>
void applicator<Iter, Func>::apply(Iter begin, Iter end)
{
   while (begin != end) {
       if (!func(*begin))
           break;
       ++begin;
   }
}
```

בנוסף, לאפליקטור יש **אופרטור קריאה לפונקציה** - **operator ()** - המאפשר להשתמש באפליקטור כזה כפונקציה לאפליקטור אחר.

כשהאפליקטור נתון לנו, אנחנו יכולים לנצל אותו כבר עתה, לשני השימושים שלמדנו. אנו יכולים, למשל, להשתמש באפליקטור למטרת הדפסת אובייקטים ברשימה, או בכל מערך אחר.

## 7.2.3.1 הדפסה בעזרת האפליקטור

כדי להשתמש באפליקטור צריך להגדיר פונקציה, השולחת אובייקטים כלשהם לפלט הסטנדרטי. אפשר גם להגדיר אובייקט שמתנהג כפונקציה, בדרך הבאה:

```
template <class T>
struct printer {
        int operator()(const T &val)
        { cout << val << " "; return 1; }
};
```

printer היא מחלקת תבנית שמדפיסה כל אובייקט לפלט הסטנדרטי. המחלקה מגדירה את אופרטור הקריאה לפונקציה ומשתמשת באופרטור הסטנדרטי (<<) לניתוב אובייקטים לזרם כלשהו. אופרטור הקריאה לפונקציה מחזיר ערך 1, כדי שהמעבר על האובייקטים בתחום הנתון יכלול את כל האובייקטים, ולא ייפסק לאחר האובייקט הראשון. עבור מחלקה כלשהי T צריך האופרטור "<<" להיות מוגדר עבור מחלקה זו, כדי שאובייקט הפונקציה printer יוכל לפעול על המחלקה.

```
int ai[]= {1,2, 3, 4, 5, 6, 7};
applicator<int*, printer<int> > prt(printer<int>());
prt(ai, ai + sizeof(ai) / sizeof(int));
```

קטע קוד זה מגדיר מערך של 7 שלמים. לאחר מכן, מוגדר אפליקטור המתאים למערך של שלמים (הפרמטר הראשון של התבנית מקבל מצביע לשלמים) ומקבל אובייקט מסוג printer. לבסוף, נעשה שימוש באפליקטור, כדי להדפיס את המערך לפלט הסטנדרטי.

אם כן, האפליקטור יכול לחסוך לנו את כל הפונקציות העוברות על תחום המוגבל בין שני איטרטורים, ומבצע פעולה על כל אובייקט בתחום.

## 7.2.3.2 פירוק בעזרת האפליקטור

באופן דומה להדפסה בעזרת האפליקטור, אפשר גם לפרק אובייקטים, הנמצאים ברשימת המצביעים. כאן נראה רק את קטעי הקוד שישתנו בתוכנית המציירת צורות, כדי לא לחזור על דברים ידועים. הקוד המלא נמצא בקובץ **polym3.cpp**. כדי לפרק את האובייקטים ברשימת הצביעים, נגדיר **אובייקט-פונקציה** (function object) שכולל את אופרטור הקריאה לפונקציה. אופרטור זה מקבל מצביע לצורה ומפעיל את המפרק שלה. האופרטור מחזיר 1 כדי שהאפליקטור יסרוק את כל האובייקטים בתחום הנתון.

```
struct shape_destroy {
      int operator()(Shape *sp) {
            sp->Shape::~Shape();
            return 1;
      }
};
```

בפונקציה הראשית נגדיר אפליקטור שהפרמטר הראשון שלו הוא איטרטור של הרשימה, והפרמטר השני הוא אובייקט הפונקציה שמפרק אובייקטים מסוג Shape. בנוסף, נעביר אובייקט המבצע את הפירוק לבנאי של האפליקטור.

```
int main()
{
      list<Shape*> shapes;
      Screen scr;
      applicator<list<Shape*>::list_iter,
               shape_destroy>applic(shape_destroy());
      shapes.append(new Line(10,1, 19, 11));
      shapes.append(new Line(0,0, 0, 10));
      shapes.append(new Line(0,0, 10, 0));
      shapes.append(new Line(10,0, 10, 10));
      shapes.append(new Line(0, 10, 10, 10));
      shapes.append(new Line(40, 20, 55, 10));
      shapes.append(new Point(10, 20));
      shapes.append(new Point(15, 10));
      draw(shapes, scr);
      applic(shapes.begin(), shapes.end());
      return 0;
}
```

כשמפעילים את האפליקטור קובעים לו תחום. במקרה זה, התחום הוא מתחילת הרשימה ועד סופה.

## 7.2.4 דוגמה - מערך דינמי

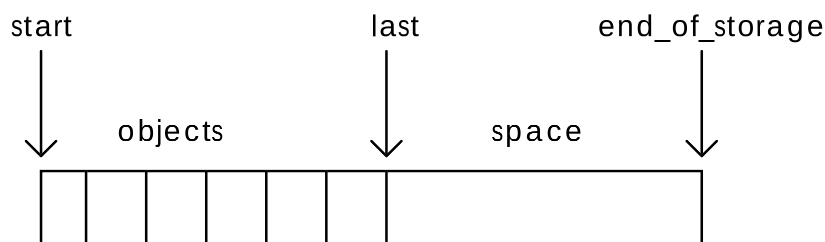
בסעיף זה נציג דוגמה נוספת למחלקת תבנית - **מערך דינמי** (dynamic array). מערך דינמי דומה למערך בסיסי של C++ ויעיל כמו מערך רגיל, אך מספק את יכולת ההגדלה באופן אוטומטי, ללא כל צורך בהתערבות המשתמש להקצאת זיכרון נוסף. הקצאת הזיכרון נעשית במחלקה עצמה, בעת הצורך.

**המערך** (Array) הוא מחלקת תבנית שמקבלת פרמטר אחד, סוג האובייקט בתבנית. למערך יש איטרטור המאפשר לסרוק את המערך.

### 7.2.4.1 מבנה המערך

למערך יש מצביע (start) לתחילת אזור הזיכרון של המערך. בחלק מהמערך כבר מאוחסנים ערכים והמצביע last מצביע למקום אחד **מעבר** לאזור התפוס (ראה ציור). לכן, הערך last_start הוא מספר האובייקטים הקיימים במערך. המצביע end_of_storage מצביע לסוף האזור שהוקצה עבור המערך.

כשהמערך מקצה זיכרון, הוא מקצה מספיק מקום לאובייקטים הקיימים במערך ובנוסף, אותה כמות זיכרון לאובייקטים שאינם מאותחלים. שיטת עבודה זו נועדה לחסוך במספר הפעמים בהן יש להקצות זיכרון.




הקצאת זיכרון אינה נעשית בעזרת האופרטור new, משום הוא גורם להפעלת בנאי האובייקטים שנמצאים במערך. דבר זה אינו מחויב המציאות, מכיון שייתכן שאין צורך בכל האובייקטים במערך. במקרה ואין צורך בכל האובייקטים, אזי הקריאה לבנאי של אובייקטים אלה היא מיותרת ותגרום לחוסר יעילות.

#### 7.2.4.1.1 יצירת אובייקטים במקום מסוים במערך

כשמוסיפים אובייקטים למערך יש לבנות אותם באזור הזיכרון שאינו מאותחל באובייקטים. כזכור, כשיש אובייקטים עם פונקציות וירטואליות, מכניס המהדר מצביע לטבלה של מצביעים לפונקציות וירטואליות. לכן, יש לטפל באתחול האובייקטים בחלק הריק של המערך, כדי לאתחל מצביעים לטבלת פונקציות, עבור אובייקטים פולימורפים.

**הבנאי** הוא המקום בו מאותחל המצביע לטבלת המצביעים של הפונקציות הווירטואליות. בכל בנאי מוסיף המהדר קוד שמאתחל את המצביע לפי סוג האובייקט הנוכחי. המתכנת אינו יכול לקרוא לבנאי של אובייקט באופן ישיר. כדי לעשות זאת, אפשר לנצל את העובדה, שכשקוראים לאופרטור new מוסיף המהדר קריאה לבנאי של האובייקט, עם הכתובת המוחזרת על ידי האופרטור **new**.

ב-C++ אפשר להגדיר אופרטור new שמחליף את אופרטור המערכת. אופרטור כזה, המקבל את כתובת ההקצאה ומחזיר את אותה הכתובת, יאפשר להפעיל את בנאי האובייקט על אותו אזור זיכרון. בכך נגרום ליצירה אוטומטית של אובייקט חדש באזור הזיכרון המבוקש. בדרך זו המהדר מעדכן את המצביע לטבלת הפונקציות בצורה אוטומטית. שיטה זו תפעל בכל מהדר C++, ובכל פלטפורמה.

נגדיר אופרטור הקצאת זיכרון new בדרך הבאה:

```
void *operator new(unsigned int, void *p) { return p; }
```

האופרטור מחזיר את הכתובת המועברת לו. עבור מחלקה כלשהי X אנו יכולים כעת להשתמש באופרטור באופן הבא:

```
char xmem[sizeof(X)];
new (xmem) X;
```

קטע קוד זה יוצר אובייקט מסוג X באזור הזיכרון xmem. באופן דומה ובעת הצורך, ניתן להפוך את הזיכרון בתחום (last, end_of_storage) שאינו מאותחל, לאובייקטים מסוג נתון.

### 7.2.4.1.2 הממשק של Array

ממשק המערך מגדיר את האיטרטור כמצביע לאובייקט שנמצא בו. מכך נובע, שסריקת מערך זה יעילה כמו סריקה של כל מערך רגיל של C++. הגדרת האיטרטור נעשית בחלקו הציבורי של המערך, וכך שאפשר להשתמש בו גם מחוץ למחלקה.

הפונקציה construct מאפשרת ליצור אובייקט במקום כלשהו במערך. היא מקבלת את המיקום והערך בהם יש לאתחל את האובייקט. הפונקציה מנצלת את האופרטור new שהגדרנו, ומשתמשת בבנאי העתקה של האובייקט כדי לאתחל אותו בערך הנתון.

```
template <class Type>
class Array {
public:
    typedef Type* iterator;
    enum { page_size = 512 };
private:
    iterator start;
    iterator last;
    iterator end_of_storage;
    void construct(iterator pos, const Type &val)
    { new(pos) Type(val); }
```

```
    void destroy(iterator i1, iterator i2);
public:
    void resize(int sz);
    Array(int sz=page_size)
    : start(0), last(0), end_of_storage(0)
    { resize(sz); }
    ~Array() { destroy(start, last); free(start); }
    Type &operator[](int i) { return *(start+i); }
    int size() const { return (last - start); }
    int capacity() const { return (end_of_storage - start);
}
    void push_back(const Type  &val);
    iterator begin() { return start; }
    iterator end()   { return last;  }
};
```

הפונקציה destroy מפרקת אובייקטים בתחום כלשהו במערך. התחום מוגדר לפונקציה על ידי שני איטרטורים, כלומר שני מצביעים. הפונקציה מפעילה באופן ישיר את המפרק של כל אובייקט בתחום הנתון. כבר ראינו שפונקציות כאלו יכולות להיות ממומשות בעזרת אפליקטור (ראה תרגילים).

```
template <class Type>
void Array<Type>::destroy(iterator i1, iterator i2)
{
    while (i1 != i2) {
         i1->Type::~Type();
         ++i1;
    }
}
```

הפונקציה resize משמשת להקצאת זיכרון. הקצאת זיכרון יכולה להתרחש בשני מקרים: כשיש צורך לאתחל את גודל המערך באופן ראשוני, או כשיש צורך להכניס אובייקט נוסף למערך, ואז יש להגדיל את מימדי המערך. בשני המקרים מבצעת הקריאה לפונקציית המערכת realloc את ההקצאה הנדרשת. כשפונקציה זו נקראת מהבנאי מאופסים המצביעים של המערך, ולכן היא תקצה את המערך. אחרת, היא תגדיל (realloc) את מימדיו.

```
template <class Type>
void Array<Type>::resize(int sz)
{
    int n = size();
    start = (iterator)realloc(start, sz * sizeof(Type));
    last = start + n;
    end_of_storage = start + sz;
}
```

**האופרטור []** מחזיר ייחוס לאלמנט של המערך. אופרטור זה מאפשר להשתמש במחלקה מערך, כמו במערך בסיסי של השפה. למשל:

```
Array<int> ai(1);
//...
ai[5] = 3;
```

באופרטור [] אין אפשרות להשתמש מייד לאחר שהמערך מוגדר, משום שבתחילה מוגדר מערך עם הקצאה מספקת (10 שלמים בדוגמה), אבל זיכרון זה אינו מאותחל לאובייקטים. לפיכך, יש תחילה להשתמש בפונקציה push_back. בתרגילים, אנחנו מוספים פונקציות המאפשרות להגדיר ערכים התחלתיים במערך.

האופרטור [] אינו בודק חריגה מתחום המערך, ולכן כל חריגה כזו יכולה לגרום לתופעות ולתוצאות שאינן מוגדרות, או צפויות. לו היינו בודקים חריגה בכל שימוש באופרטור, היה השימוש בו גורם להפחתת היעילות בצורה משמעותית. בהמשך, נראה פתרונות אחרים לבעיה.

הפונקציה push_back כותבת אובייקט לסוף המערך. אם יש צורך (לדוגמה, במקרה שאין מספיק מקום במערך) מוכפל גודלו של המערך. לאחר מכן, מאותחל האובייקט בכתובת של last ומצביע זה מוגדל בהתאם.

```
template <class Type>
void Array<Type>::push_back(const Type &val)
{
   if (last >= end_of_storage)
       resize(capacity() * 2 + 1);
   construct(last, val);
   last++;
}
```

### 7.2.4.1.3 שימוש במערך

עד כה ראינו את המחלקה עצמה וכעת נראה כיצד אפשר להשתמש בה. כדי לבחון את המערך ולבנות אובייקטים במקום בו יש זיכרון לא מאותחל, נגדיר אובייקט עם פונקציה וירטואלית. אובייקט זה מייצג שלם, ולכן נוסיף לו פונקציה וירטואלית שתחזיר את ערך השלם שהוא מכיל.

```
class Int {
    int val;
public:
    Int(int v) { val = v; }
    virtual ~Int() {}
    virtual int get() const { return val; }
};
```

כדי לאפשר מיון של המערך נוסיף שני אובייקטי פונקציות. הראשון, מופע של מחלקת התבנית less עבור אובייקטים מסוג Int. באותו אופן, נגדיר גם **אובייקט-פונקציה** שהוא מופע של מחלקת התבנית great. האובייקט הראשון הוא לטובת אלגוריתם המיון quick_sort, והשני לטובת האלגוריתם insertion_sort.

```
struct less<Int> {
     int operator()(const Int &i1, const Int &i2) const
     { return (i1.get() < i2.get()); }
};
struct great<Int> {
     int operator()(const Int &i1, const Int &i2) const
     { return (i1.get() > i2.get()); }
};
```

כדי להשתמש ב**אפליקטור** ולהדפיס את המערך נגדיר אובייקט-פונקציה עם אופרטור קריאה לפונקציה, המדפיס אובייקט לפלט הסטנדרטי תוך שימוש באופרטור "<<". כדי להדפיס אובייקטים מסוג Int נגדיר אופרטור זה עבור אובייקטים מתאימים.

```
template <class T>
struct printer {
        int operator()(const T &val)
        { cout << val << " "; return 1; }
};

ostream &operator<<(ostream &out, const Int &i)
{ return out << i.get(); }
```

נגדיר שתי פונקציות שימלאו מערכים המכילים אובייקטים מסוג int ואובייקטים מסוג Int. פונקציות אלו משתמשות בפונקציה push_back, הכותבת אובייקט בסוף המערך, ובעת הצורך מקצה למערך זיכרון נוסף.

```
void fill_array(Array<int> &a, int n)
{
    a.resize(n/2);
    for (int i=n; i>0; i--)
       a.push_back(i);
}

void fill_array(Array<Int> &a, int n)
{
    a.resize(n/2);
    for (int i=n; i>0; i--) {
       Int k(i);
       a.push_back(k);
    }
}
```

הפונקציה הראשית משתמשת בפונקציות שלמדנו בפרק זה, כדי למלא שני מערכים. הראשון מביניהם הוא מערך של שלמים והשני, מערך של אובייקטים מסוג Int. המערך הראשון ממוין בעזרת קריאה לפונקציה quick_sort. לאחר מכן מודפס המערך בעזרת קריאה לאלגוריתם הגנרי print. קריאה לפונקציה reverse הופכת את סדר האובייקטים במערך. לאחר הפעלת הפונקציה האחרונה מסודרים האובייקטים מאובייקט גדול לקטן.

את אפקט סידור האובייקטים בסדר יורד יכולנו להשיג אם היינו מעבירים לפונקציית המיון quick_sort אובייקט פונקציה מסוג great, המתאים לסוג האובייקטים במערך. דבר זה יעיל יותר, שכן היינו חוסכים מעבר אחד על המערך.

פעולות דומות מופעלות גם על המערך השני. הפלט של התוכנית הבאה, קובץ **tmpl1.cpp**, הוא שתי הדפסות של המערך הראשון ושתי הדפסות של המערך השני.

```
int main()
{
    Array<int> ai;
    Array<Int> a1;
    applicator<int*, printer<int> >
    prt(printer<int>());
    fill_array(ai, 30);
    print(ai.begin(), ai.end());
    quick_sort(ai.begin(),  ai.end(), less<int>());
    print(ai.begin(), ai.end());
    reverse(ai.begin(), ai.end());
    cout << "after reverse:" << endl;
    prt(ai.begin(), ai.end());
    cout << "---------------------------------" << endl;
    fill_array(a1, 30);
    print(a1.begin(), a1.end());
    Int tmp(0);
    insertion_sort(a1.begin(),a1.end(),great<Int>(),tmp);
   print(a1.begin(), a1.end());
   return 1;
}
```

בדוגמה זו ראינו כיצד מפעילים אלגוריתמים גנרים על מבני אלגוריתמים שונים, ללא כל צורך בשכתוב מחדש של האלגוריתמים.

## 7.3 ירושה ותבניות

בפרק זה ראינו שאפשר להשתמש במנגנון הירושה עבור מחלקות תבנית. המחלקה list ירשה ממחלקה אחרת (base_list) שביצעה את מירב העבודה, כדי למנוע ניפוח של הקוד. יש אפשרות מורכבת אף יותר לרשת מחלקת תבנית ממחלקת תבנית אחרת.

עובדה זו מקנה מימד חדש ועוצמה רבה למחלקות תבנית. **מחלקות תבנית אינן מאקרו**, והן מאפשרות להשתמש בירושה ופולימורפיזם כדי להשיג גמישות גדולה יותר.

## 7.3.1 ירושה מהמחלקה Array

המחלקה Array שראינו בסעיף הקודם אפשרה לגשת לאובייקטים במערך בעזרת אופרטור []. לכן, השימוש באובייקטים של מחלקה זו דומה מאוד לאופן השימוש במערכים בסיסיים של השפה. כמו במערכים בסיסיים של השפה, גם כאן אין הגנה מפניות אל מחוץ לגבולות המערך.

כדי לטפל בנושא זה ניצור מחלקת תבנית חדשה, היורשת ממחלקת התבנית הקודמת, ומבצעת בדיקות חוקיות על פניות לאינדקסים במערך.

המחלקה SafeArray היא מחלקת תבנית היורשת מהמחלקה Array ומגדירה בדיקת תחום בכל פנייה לאופרטור []. מאחר והמחלקה SafeArray היא מחלקת תבנית עם פרמטר T, הרי שהמחלקה הבסיסית שלה היא המחלקה Array<T>, כלומר, מחלקת תבנית עם אותו הפרמטר. הקוד של מחלקה זו נמצא בקובץ **SafeArray.H**.

כדי להחזיר ערך כלשהו עבור פניה לאינדקס שמחוץ למערך מוסיפים שדה nil. שדה זה הוא **סטטי** (static), כלומר, קיים רק פעם אחת עבור המחלקה ומשותף לכל האובייקטים שלה. כשיש הגדרה כזו עבור מחלקה רגילה, אנו מבינים את משמעותה. משמעות ההגדרה שונה במקרה של מחלקת תבנית.

כשיש לנו הגדרה של **שדה סטטי** במחלקת תבנית, משותף שדה זה לכל האובייקטים של המחלקה **עבור אותם פרמטרים** של התבנית. עבור פרמטר תבנית מסוג int יש אובייקט אחד בלבד המייצג את השדה nil. עבור פרמטר אחר, מסוג double, יהיה אובייקט סטטי אחר, מסוג double, המייצג את השדה הסטטי. כלומר:

```
SafeArray<double> sa1(5), sa2(10);
SafeArray<int>    sa3(4), sa4(12);
```

לאובייקטים sa1 ו-sa2 יש שדות משותפים עבור המשתנה הסטטי, השונה מהמשתנה הסטטי של האובייקטים sa3 ו-sa4. לשניים האחרונים יש את אותו אובייקט nil.

```
template <class T>
class SafeArray : public Array<T> {
        static T nil;
public:
        SafeArray(int sz = page_size) : Array<T>(sz)
        {}
        T &operator[](int i);
        static const T &get_nil()
        { return nil; }
};
```

האופרטור [] מוגדר מחדש עבור המחלקה SafeArray. אופרטור זה, במחלקה זו, מבצע בדיקה על האינדקס הנתון לו. אם האינדקס נמצא בתחום הנוכחי של המערך, מוחזר אובייקט מתאים, בעזרת קריאה לאופרטור [] של המחלקה הבסיסית. אחרת, האינדקס מחוץ לתחום התקין של המערך, ומוחזר ייחוס לאובייקט הסטטי.

```
// use default constructor of T for static nil
template <class T> T SafeArray<T>::nil;

template <class T>
T &SafeArray<T>::operator[](int i)
{
     if (i>=0 && i<size())
            return Array<T>::operator[](i);
     return nil;
}
```

יש שני חסרונות בגישה זו. ראשית, הגדרת אובייקט סטטי nil המשתמש בבנאי של ברירת המחדל שלו. דבר זה דורש שאובייקטים מסוג T יכללו בנאי של ברירת המחדל. חיסרון שני הוא טיפול במצב שגיאה בעזרת החזרת ייחוס לאובייקט מסוים. למשתמש בקוד זה אין דרך לדעת שהוחזר אובייקט סטטי כזה. בהמשך נראה שיטות טובות יותר לטיפול במצבי שגיאה.

## 7.3.2 שימוש באובייקט מסוג מערך בטוח

השימוש במערך בטוח כזה דומה לשימוש במערך רגיל, ונראה כך :

```
SafeArray<int> sa;
sa.resize(10);
for (int i=0; i<10; i++)
     sa.push_back(i+1);
for (i=0; i<15; i++)
     cout << sa[i] << " ";
cout << endl;
```

קטע קוד זה מגדיר מערך בטוח בגודל של 10 אובייקטים ומכניס 10 אובייקטים לתוכו. לאחר מכן מוסיפים 15 אובייקטים. הפלט של קטע הקוד הבא הוא :

```
1 2 3 4 5 6 7 8 9 10 0 0 0 0 0
```

כלומר, עבור האובייקטים החורגים מהמערך, מודפס ערך של השדה nil, שהוא אפס.

### 7.3.2.1 פונקציות וירטואליות

הגדרת האופרטור [] במחלקה Array כפונקציה וירטואלית היתה מאפשרת לכתוב פונקציות המנצלות זאת, באופן הבא :

```
void draw_shapes(Array<Shape*> &arr, Screen &scr)
{
    int i = 0;
    while (arr[i] && i<arr.size()) {
      arr[i]->draw(scr);
      i++;
    }
}
```

כשהשימוש באופרטור [] הוא פולימורפי, בהתאם לסוג האובייקט האמיתי של המערך. אכן, דבר זה מקנה גמישות נוספת למחלקה זו, אך עבור גמישות יש לשלם. התשלום במקרה זה, הוא הפעלה פולימורפית של אופרטור זה, שהיא פחות יעילה מהפעלה רגילה. המחיר אינו גדול, אבל אם הפנייה לאופרטור זה נעשית בלולאה הפנימית ביותר, הופך התשלום להיות משמעותי יותר.

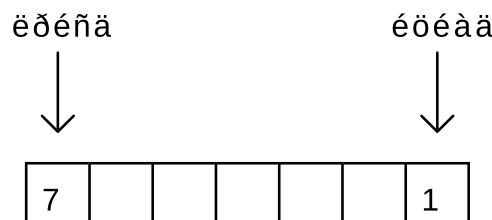
## 7.3.3 מתאמים

**מתאמים** (adaptors) הם מחלקות המשתמשות במחלקות אחרות לצורך מימוש פונקציונליות מסוימת. המחלקה המתאמת משתמשת במחלקה אחרת למימוש הפונקציונליות שלה, והפונקציות שלה קוראות לפונקציות של המחלקה בה משתמש המתאם.

כדי להדגים מושג זה נפתח שתי מחלקות, המייצגות תור (queue) ומחסנית (stack) בעזרת המחלקה רשימה (list).

### 7.3.3.1 תור

**תור** (queue) הוא מבנה נתונים המכיל אובייקטים אחרים וגורם לסדר מסוים ביניהם. התור פועל בשיטת **FIFO** (קיצור ל-First In Fist Out). כלומר, האלמנט הראשון שמוכנס לתור הוא גם האלמנט הראשון שיוצא ממנו.




באיור רואים שהאובייקט הראשון שנכנס כבר קרוב ליציאה, והאחרון יהיה גם האחרון שיוצא מהתור.

כדי ליצור תור, נגזור בצורה פרטית את המחלקה queue מהמחלקה list :

```
template <class T>
class queue : private list<T> {
public:
    queue() {}
    void put(const T &val) { insert(val); }
    T get() { return remove_tail(); }
    int size() const { return list<T>::size(); }
};
```

מתאמים מבצעים תיאום של ממשק מחלקה אחת לממשק אחר. במקרה זה, התיאום המתבצע הוא של הרשימה לתור. במחלקה queue נפרשות כל הפונקציות על ידי המהדר, ולכן, הוא לא מייצר קוד כלשהו.

הירושה הפרטית ששימשה אותנו במקרה זה גורמת לכך שהמשתמש במחלקה זו אינו יכול לגשת לפונקציות המחלקה הבסיסית. אם היינו משתמשים בירושה ציבורית, יכול היה המשתמש במחלקה לגשת לפונקציות אחרות של הרשימה, ובכך להפר את תנאי התור. בצורה שקולה לירושה פרטית, יכולנו להגדיר את הרשימה כשדה בתור.

השימוש במחלקה זו נראה בצורה הבאה :

```
queue<int> qi;

for (int i=0; i<10; i++)
    qi.put(i);

while (qi.size() > 0)
    cout << qi.get() << " ";
```

קטע קוד זה גורם לפלט הבא :

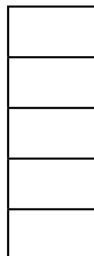
```
0 1 2 3 4 5 6 7 8 9
```

## 7.3.3.2 מחסנית

**מחסנית** (stack) היא מבנה נתונים המכיל אובייקטים אחרים. המחסנית מאפשרת הכנסה והוצאה של אובייקטים תוך שמירה על סדר מסוים ביניהם. האובייקט שמוכנס ראשון למחסנית הוא גם זה היוצא ממנה ראשון, כלומר, היא פועלת בשיטת **LIFO** (כמו מחסנית של רובה).

כדי לממש את המחסנית נגדיר מחלקת תבנית שמכילה רשימה ומתאמת את ממשק הרשימה לממשק המבוקש של מחסנית.

äëðñä ↓ ↑ äåöàä



```
template <class T>
class stack {
    list<T> rep;   // representation
public:
    stack() {}
    void push(const T &val)
    { rep.insert(val); }
    T pop()
    { return rep.remove_head(); }
    int size() const { return rep.size(); }
};
```

במקרה זה יכולנו להשתמש גם בירושה פרטית (השקולה להכלה) אבל, כדי להדגים את האפשרות השנייה מימשנו את המחסנית בעזרת הכלה.

אפשר לראות שהמחסנית מגדירה פונקציה אחת (push) להכנסה של אובייקט לתוך המחסנית ופונקציה נוספת (pop) להוצאת אובייקט מהמחסנית. השימוש במחלקה מחסנית הוא בדרך הבאה:

```
stack<int> si;
for (int i=0; i<10; i++)
    si.push(i);
while (si.size() > 0)
    cout << si.pop() << " ";
```

קטע הקוד הבא ידפיס את הפלט הבא לפלט הסטנדרטי:

```
9 8 7 6 5 4 3 2 1 0
```

### 7.3.3.3 מתאמים גנרים

עד כה, ראינו מתאמים המשתמשים במחלקה מסוימת כדי לממש תור, או מחסנית. פתרון גנרי טוב יותר הוא לאפשר תור, או מחסנית, כשהמימוש נקבע על ידי המשתמש, בעת הגדרת המחלקה. לדוגמה:

```
template <class Container>
class queue {
    Container cont;
public:
    queue() {}
    void put(const value_type &v)
    { cont.insert(v); }
    value_type get()
    { return cont.remove_head(); }
};
```

בקטע קוד זה מקבלים, כפרמטר של תבנית, את סוג המכולה (container) בה יש להשתמש כדי לממש את התור. השימוש במחלקה זו נראה כך:

```
queue<list<int> > qi;
qi.put(1);
int i = qi.get();
```

הבעיה בקטע קוד זה היא שאין לנו את סוג האובייקטים המוכנסים ומוצאים מהתור בהגדרת מחלקת התבנית. קטע קוד זה לא יעבור את שלב ההידור. אפשרות אחת היא להוסיף סוג זה כפרמטר של התבנית:

```
template <class Container, class T>
class queue {
...
};

queue<list<int>, int> qi;
...
```

גישה זו אמנם פותרת את הבעיה, אך פחות נוחה למשתמש, שצריך להגדיר פעמיים את סוג האובייקטים במכולה. כדי לפתור בעיה זו אפשר להגדיר את סוג האובייקטים במכולות בדרך הבאה:

```
template <class T>
class list {
...
public:
    typedef T value_type;
    ...
};
```

לאחר שהגדרנו את סוג האובייקטים במכולה, אפשר להגדיר את המתאם בדרך זו:

```
template <class Container>
class queue {
    typedef Container::value_type value_type;
    Container cont;
    queue() {}
    void put(const value_type &v)
    { cont.insert(v); }
    value_type get()
    { return cont.remove_head(); }
};

queue< list<int> > qi;
```

כעת, השימוש הרבה יותר נוח למשתמש. הוא יכול להחליט מה סוג האובייקט והמכולה שתשמש למימוש התור, ללא כל צורך בחזרות על סוג האובייקט. בגישה זו, על כל המכולות להגדיר את סוג האובייקט, כך שאפשר יהיה להשתמש בהם עבור מתאמים.

## 7.4 סיכום

בפרק זה הכרנו את המושג **תבנית** ב-++C. יש שני סוגי תבניות: סוג אחד הוא **פונקציות תבנית** והסוג השני הוא **מחלקת תבנית**. קיימת אפשרות לשילוב בין פונקציות תבנית גנריות לאובייקטים השייכים למחלקות תבנית.

**פונקציות תבנית** פועלות על סוגי הנתונים כפרמטר של התבנית. היכולת להגדיר פונקציות כאלו, המסוגלות לפעול על סוגים שונים של פרמטרים, מאפשרת לפתח פונקציות גנריות הטובות למספר רב של סוגי אובייקטים. פיתוח פונקציה גנרית כזו הוא קל. ראשית, מפתחים פונקציה רגילה, שאינה פונקציית תבנית, ופועלת על סוג מסוים. לאחר שהפונקציה פועלת כשורה, אפשר להפוך אותה לפונקציית תבנית גנרית.

**מחלקות תבנית** דרושות כשיש מחלקה המבצעת פונקציונליות שאינה תלויה בסוג האובייקטים בהם היא מטפלת. דוגמה לכך היא אוספים, או מכולות. מחלקות מסוג זה מכילות אובייקטים אחרים, במבנה נתונים מסוים. מחלקות אלו מאפשרות לסרוק את המכולה, להוסיף או לבטל בה אובייקטים בכל עת.

אפשר להשתמש ב**ירושה** וב**פונקציות וירטואליות** כשמשתמשים במחלקות תבנית. בכך שונות מחלקות תבנית ממאקרוס פשוטים של השפה. השימוש בירושה ובפונקציות וירטואליות, מקנה מימד חדש ועוצמה רבה ל-++C.

## 7.5 שאלות

1. כיצד אפשר לסדר שהפונקציות print ו-quick_sort יהיו בעלות אותו ממשק?
2. האם יש דרך לבטל את הפרמטר tmp (האחרון) המועבר לאלגוריתם insertion_sort?
3. האם אפשר להפעיל את האלגוריתם quick_sort על הרשימה? ואם לא, מה יש לעשות כדי לאפשר זאת?
4. הגדר רשימה של מצביעים גנרים (*void) בעזרת רשימת התבנית. גזור מחלקת תבנית חדשה מרשימת המצביעים הגנרים. ממש את דוגמת הצורות בעזרת רשימה זו.
5. שנה את המחלקה printer כך שאפשר יהיה לשלוח כל אובייקט לכל קובץ נתון.
6. הוסף פונקציה remove למערך (Array). על הפונקציה לבטל אובייקט במערך. כיצד מפרקים אובייקט כזה?
7. הוסף פונקציה שמכניסה אובייקטים לאמצע המערך (Array). מהי הסיבוכיות של פונקציה זו?
8. הוסף פונקציה המכניסה מספר אובייקטים לכל מקום במערך בצורה רצופה. מהי סיבוכיות פונקציה זו? האם עליך להשתמש בפונקציה הקודמת? האם זה יעיל להשתמש בפונקציה הקודמת?
9. שנה את הפונקציה destroy של המחלקה Array כך שתשתמש באפליקטור.
10. הוסף בנאי המאפשר לאתחל מערך ממערך אחר בסיסי של ++C, שנראה כך:

```
Array(const T *arr, int n);
```

כאשר n הוא מספר האלמנטים הנמצאים במערך arr.

פרק 8

# טיפול בשגיאות ועוד

בפרק זה נלמד על הדרך בה מטפלים בשגיאות ב-++C. טיפול בשגיאות נתמך בשפת התכנות על ידי מבנים כגון: **try**, **throw**, **catch**. התמיכה למבנים אלה בשפת ++C מוסיפה לה מורכבות מיותרת, במיוחד כשמשתמשים בתבניות. נלמד כיצד לגלות בעיות אלו וכיצד לפתור אותן.

בפרק זה נפתֵח כלים ומחלקות שיעזרו לנו להתמודד עם מצבים חריגים. מרבית המהדרים אינם תומכים במצבים חריגים, וגם אלה שתומכים בהם, עדיין אינם תומכים בכל האפשרויות שמאפשר התקן של שפת ++C. **כדי להריץ את הדוגמאות בפרק זה, עליך להשתמש במהדר התומך במצבים חריגים.**

## 8.1 מדוע צריך לטפל בשגיאות

מה החשיבות של טיפול בשגיאות? האם אי אפשר להסתפק בהחזרת הודעה, או התרעה על שגיאה כתוצאה מקריאה לפונקציה, למשל? הנה דוגמה:

```
// return a file descriptor or -1 on error
int open_file(const char *name)
{
      //...
}
```

אכן, בשיטה זו נוהגים זמן רב: המשתמש בפונקציה צריך לבדוק בעצמו את הערך המוחזר על ידי הפונקציה. לכן, התוכנה של המשתמש בפונקציה זו, תיראה כך:

```
int fd = open_file("db");
if (fd == -1)
// ... handle error
```

ולמרות זאת, יש בכך חיסרון. קוד המשתמש משובץ בבדיקות שגיאה בכל מקום, דבר שמתאפשר בכלים התקניים והמקובלים בשפת C, ובוודאי שב-++C.

ל-C++ יש מחלקות ואופרטורים המדמים מחלקה לאובייקטים בסיסיים של השפה. לדוגמה:

```
class String {
      char *str;
public:
      String(const char *s);
      //...
      char &operator[](int i);
      //...
      int bad() const { return (str == 0); }
};
```

**בנאי אובייקט** זה מקצה זיכרון ומעתיק לתוך אזור הזיכרון החדש את המחרוזת הנתונה לו.

```
String::String(const char *s)
{
    str = new char[strlen(s) + 1];
      // what if no memory is available?
    if (str)
      strcpy(str, s);
}
```

מה קורה אם אופרטור ההקצאה אינו מצליח להקצות זיכרון כנדרש? במקרה כזה האובייקט לא יכול להקצות זיכרון, ולכן נווצר מצב לא תקין. הבנאי אינו יכול להחזיר ערך שמראה על בעיה. לשם כך, יש להוסיף פונקציה שמחזירה ערך אם האובייקט תקין, ואז קוד המשתמש ייראה כך:

```
String s("This is a long string");
if (s.bad()) {
      // error handling...
}
```

בעיה אחרת היא השימוש באופרטור []. אם פונים לאזור שנמצא מחוץ למחרוזת, מה עלינו לעשות? האם להחזיר תו מיוחד? במקרה כזה, איך נבדיל בין התו המיוחד לשאר התווים? בעיה זו יכולה להיות בעבודה עם כל אופרטור, כמו אופרטור השמה, למשל.

```
String &String::operator=(const String &rhs)
{
      delete [] str;
      str = new char[strlen(rhs.str) + 1];
      if (!str) {
            // handle error ?
      }
      strcpy(str, rhs.str);
      return *this;
}
```

גם באופרטור זה, אין לנו אפשרות להחזיר ערך המסמן שגיאה. לכן, חייב קוד המשתמש לבדוק את האובייקט **לאחר** השמה, כך:

```
String s1("s1"), s2("s2");

s1 = s2;
if (s1.bad()) {
      // ... handle error
}
```

לסיכום, קיימות מספר בעיות:

- ב-C++ יש פונקציות, כגון בנאים או אופרטורים, שאינם מחזירים הודעות שגיאה.
- כשמקבלים הודעות כאלו, יש לבדוק בכל מקום בנפרד, מה הסיבה.

## 8.2 פונקציות טיפול בשגיאות

אפשר להעביר לתוכנית פונקציה שתיקרא במקרה שגיאה. למשל, לאופרטור new היתה אפשרות לקרוא לפונקצית טיפול בשגיאה, כשאין די זיכרון להקצות. במקרים כאלה גישה זו יעילה, ובמיוחד כשיש אפשרות להתגבר על בעיה זו. למשל:

```
void (*new_handler_ptr)();

void *operator new(size_t sz)
{
      void *p = malloc(sz);
      if (p == 0 && new_handler_ptr) {
            (*new_handler_ptr)();
            p = malloc(sz);
      }
      return p;
}
```

בדוגמה זו של האופרטור new, מוקצה זיכרון באמצעות פונקציית המערכת malloc. האופרטור קורא לפונקצייה המטפלת במצב שבו אין מספיק זיכרון פנוי להקצאה. במצב זה ניתנת האפשרות לשחרר זיכרון ולכן, לאחר הקריאה לפונקציית השגיאה, נעשה ניסיון נוסף להקצאת הזיכרון. אם שגרת הטיפול במצב של חוסר זיכרון **מצליחה** לשחרר מספיק זיכרון, תחזיר הקריאה הבאה ל-malloc מצביע לאזור זה ולא תיכשל שנית.

כאן יש ניסיון להתגבר על הבעיה ולהקצות זיכרון בכל זאת. אך, כשאין זיכרון לאחר הניסיון השני, חוזרים למצב הקודם, שיש לבדוק שוב את המצביע המוחזר מהאופרטור new.

## 8.3 טיפול בסיסי בעזרת מבני ++C

ב-++C יש תמיכה לטיפול בשגיאות. תמיכה זו מורכבת ממספר מבנים חדשים שהתוספו לשפה. ב-++C ניתן להעביר מידע מנקודת השגיאה אל פונקציה כלשהי המטפלת בשגיאה. מידע זה אינו מוגבל, ויכול גם להיות אובייקט שלם. יש שלושה היבטים לטיפול בשגיאות ב-++C: **try**, **throw** ו-**catch**. בסעיף זה נראה את הבסיס והתחביר של מבנים אלה.

### 8.3.1 לעורר שגיאה (Throwing an Error)

כשמתרחשת שגיאה יש להעביר את המידע אודותיה אל התוכנית המטפלת במצב החדש שנוצר. הפעולה של זיהוי השגיאה והעברתה כמצב חריג לתוכנית המטפלת בכך נקרא **עירור מצב שגיאה** (Throw an Error). עושים זאת בעזרת מילת המפתח **throw** אשר השימוש בה גורם להעברת השליטה מהתוכנית אל הפונקציה שמטפלת במצבי שגיאה.

```
void *alloc(int sz)
{
    void *ptr = malloc(sz);
    if (!ptr)
       throw("No memory available");
    return ptr;
}
```

בדוגמה זו, הפונקציה alloc מעוררת שגיאה, על ידי משפט **throw** ומעבירה מחרוזת לפונקציה המטפלת בשגיאות. פונקציה זו צריכה לשחרר זיכרון (מכיון שזו הבעיה שדווחה) ולהחזיר את השליטה לתוכנית כדי שתנסה שוב להקצות את הזיכרון הדרוש לה. לאחר מכן, נראה כיצד מגדירים שגרת טיפול בשגיאה, וכיצד מגדירים בלוק טיפול בשגיאות.

הפונקציה המעוררת שגיאה יכולה להעביר כל מידע בשעה שהיא מעוררת שגיאה. למשל, אפשר להעביר אובייקט שלם מנקודת השגיאה ונקודת העירור אל נקודת הטיפול.

```
void *alloc(int sz)
{
    void *ptr = malloc(sz);
    if (!ptr)
       throw String("No memory avilable");
    return ptr;
}
```

מכיון שאפשר להעביר אובייקטים לנקודת הטיפול בשגיאה, הרי שאפשר להעביר יותר מידע לשגרת הטיפול בשגיאה, בתקווה שהמידע יסייע לפתור את הבעיה.

## 8.3.2 אופן הטיפול בשגיאה

הטיפול בשגיאה מורכב משני חלקים. ראשית, יש להשתמש במילת המפתח **try**. כשמשתמשים במילת מפתח זו, אנו מודיעים למהדר שאנו מוכנים לטפל בשגיאות, ואז גם מספקים קטעי קוד המטפלים בשגיאות. למשל:

```
void f()
{
    try {
       // calls to other functions
    }
    catch (String &s) {
       // .. handler of error
    }
}
```

הפונקציה f משתמשת בפונקציות רבות אחרות, שאף הן עשויות להשתמש בפונקציות נוספות. פונקציות אלו יכולות לעורר מצבי שגיאה, או שפונקציות הנקראות על ידן יכולות לעורר מצבי שגיאה. מילת המפתח **try** מראה שפונקציה זו תספק קטע קוד שמוכן לטפל במצבי שגיאה. קטעי קוד אלה נמצאים בקבוצות קוד (בלוקים) שמופיעים לאחר מכן ונפתחים במילת המפתח **catch**.

יכולים להיות מספר קטעי קוד המטפלים בשגיאה, כשכל אחד מהם מטפל בסוג שגיאה אחר. כל קטע קוד מגדיר **שִגרת טיפול שגיאה** (error handler), אשר עוסקת בסוג שגיאה מסוים. המונח (המקוצר) "**שִגרת טיפול**" (או **פונקציית טיפול**) מתייחס, למעשה, לבלוק תכנות שמטפל בשגיאות ודומה במבנה שלו לפונקציה, אך **אין** זו פונקציה רגילה ב-++C.

כשקורית שגיאה באחת הפונקציות שנקראו על ידי הפונקציה f (ובתנאי שהפונקציה נקראת מתוך בלוק try) מנסה המערכת למצוא שגרת טיפול מתאימה במסגרת הפונקציה הנוכחית. אם נמצאת שגרת טיפול כזו, מועברת בקרת התוכנית אליה. אחרת, מנסה המערכת למצוא שגרת טיפול, בפונקציה שקראה לפונקציה הנוכחית (ובתנאי שזה היה מתוך בלוק **try**).

```
void f()
{
    try {
       f1();
       f2();
       // ...
    }
    catch (Cond1 &) {
       // ...
    }
    catch (Cond2 &) {
       // ...
```

```
    }
    // (1)... resume here
}
```

כאשר נמצאת שגרת טיפול במצב חריג, מבוצע קטע הקוד המתאים והבקרה מועברת לאחר כל שגרות הטיפול שבגוף הפונקציה. בקטע קוד זה תעבור הבקרה לשורה המסומנת ב-(1). כלומר, אם התעורר מצב שגיאה Cond1 בפונקציה f1, לא תבוצע הפונקציה f2. יתר על כן, פונקציה f1 תעביר את הבקרה לשגרת הטיפול (הראשונה) שמטפלת ב-Cond1, לאחר ביצוע קטע קוד זה תועבר הבקרה לשורה (1).

שגרת הטיפול בשגיאה מקבלת פרמטר בדומה לכל פונקציה אחרת. הפרמטר המועבר לשגרת הטיפול יכול להיות מצביע, ייחוס או כל סוג אחר הנתמך בפונקציות רגילות. נחזור לנושא מאוחר יותר בפרק זה.

במהלך העברת הבקרה מפונקציה לבלוק טיפול בשגיאה, דואגת המערכת לפירוק כל האובייקטים בבלוקים המופסקים. הדבר נעשה בדרך דומה לזו המתבצעת אם התוכנית היתה מגיעה לסוף הבלוק. עובדה זו חשובה במיוחד למערכות המטפלות באובייקטים, משום שיש לבצע את המפרקים של האובייקטים, שבדרך כלל משחררים משאבים כמו זיכרון, או סוגרים קבצים. בכל מקרה שלא מבצעים את המפרקים, עלולה להיות זליגת זיכרון. כלומר, זיכרון שהוקצה ולא שוחרר למערכת, וכך הוא "אובד", כי אי אפשר לנצל אותו.

## 8.3.3 דוגמה - מערך בטוח

כדי להמחיש את שלמדנו עד כה, נראה דוגמה של מערך בטוח. **מערך בטוח** (safe array) הוא מחלקה ב-C++ המספקת את הגישה לאברי המערך כמו במערך בסיסי של השפה. מערך בטוח מוסיף את בדיקת הפנייה לאינדקס המערך - **בדיקת טווח** (range check). בעת פנייה לאינדקס מחוץ לתחום הנוכחי של המערך, יעורר המערך מצב שגיאה המורה על שגיאה זו.

```
template <class T>
class SafeArray {
     T *arr;
     int size;
public:
     struct RangeError {
           int size;
           int bad_index;
           RangeError(int sz, int idx)
           : size(sz), bad_index(idx)
           {}
     };
     SafeArray(int sz) { arr = new T[size=sz]; }
     ~SafeArray() { delete [] arr; }
```

```
     T &operator[](int i)
     { if (i>=size || i<0) throw RangeError(size, i);
       return arr[i];
     }
};
```

המחלקה SafeArray היא מחלקת תבנית שהפרמטר שלה הוא סוג האובייקט במערך. המחלקה שומרת מצביע לתחילת המערך של האובייקטים בזיכרון, ואת גודלו הנוכחי של המערך.

בחלקו הציבורי של המערך מוגדרת רשומה (struct) המכילה את המידע המורה על שגיאה בפנייה לאינדקס המערך. הרשומה מכילה גם את גודלו הנוכחי של המערך וגם את האינדקס שגרם לשגיאה. הגדרת הרשומה בחלק הציבורי נועדה למנוע זיהום של מרחב השמות הגלובלי של התוכנית. חלק אחר של התוכנית יכול להגדיר RangeError ובצורה זו לא תהיה התנגשות של השמות.

הפונקציה שמעוררת שגיאה היא האופרטור של אינדקס במערך []. אופרטור זה מחזיר ייחוס לאובייקט מסוג T. אפשר להשתמש באופרטור גם לצורך השמת ערכים למערך. מכיון שלאובייקט מוחזר ייחוס מסוג T, אין כל צורה להורות על שגיאה. אופרטור זה מעורר מצב שגיאה מסוג RangeError. במצב השגיאה מוגדרים כל הפרמטרים של השגיאה שהתרחשה. אופן השימוש בקטעי הקוד הקודמים הוא כזה:

```
void fill_array(SafeArray<int> &sa)
{
    for (int i=0; i<20; i++)
          sa[i] = i;
}

int main()
{
    SafeArray<int> sa(10);
    try {
          fill_array(sa);
    }
    catch (SafeArray<int>::RangeError &re) {
    cout << "caught range error size:" << re.size
         << ", bad_index:" << re.bad_index << endl;
    }
    return 0;
}
```

הפונקציה הראשית מגדירה מערך בטוח מסוג שלם. לאחר מכן, מופיע בלוק השימוש בפונקציה שממלאת את המערך (fill_array) ב-20 ערכים. צמוד לבלוק השימוש (try) מופיע בלוק הטיפול בשגיאה אפשרית. בלוק הטיפול (catch) מטפל בשגיאות חריגה מתחום המערך.

מכיון שגודל המערך הוא 10 והפונקציה fill_array מנסה להכניס עשרים ערכים לתוכו, יש חריגה מתחום המערך. האופרטור [] מעורר מצב שגיאה כשהפונקציה fill_array פונה לאינדקס 10. הפונקציה fill_array מופסקת והבקרה מועברת לבלוק הטיפול במצב שגיאה זה. פלט תוכנית זו ייראה כך:

```
caught range error size:10, bad_index:10
```

## 8.3.4 חריגים וירושה

שפת התכנות C++ תומכת בירושה גם בעת הטיפול במצבים חריגים. המשמעות של ירושה במקרה זה היא ששגרת הטיפול במצב חריג יכולה לקבל מצביע, או ייחוס לאובייקט. על משתנים אלה פועלים החוקים הרגילים של C++. אם יש פונקציות וירטואליות, אזי הפעלה של פונקציה כזו תפעיל את הפונקציה המתאימה לאובייקט הנוכחי, שעליו מצביע המצביע.

### 8.3.4.1 ירושה ללא פונקציות וירטואליות

ניתן להשתמש בירושה עם פונקציות וירטואליות וללא פונקציות וירטואליות. בסעיף זה נראה כיצד משתמשים במנגנון השגיאות ללא פונקציות וירטואליות.

בדוגמה הקודמת ראינו רשומה שהכילה מידע על סוג השגיאה. אם נגדיר מחלקה בסיסית למצבי שגיאה, שתחזיק תיאור (במחרוזת) של הבעיה שהתרחשה, וממנה נגזור את כל המחלקות המתארות מצבי שגיאה, נקבל את הקוד הבא:

```
class BaseException {
    char *str;
    void copy(const char *s);
public:
    BaseException(const char *s) : str(0)
    { copy(s); }
    BaseException(const BaseException &be) {
         str = 0;
         copy(be.str);
    }
    BaseException &operator=(const BaseException &be) {
         if (this != &be)
                 copy(be.str);
         return *this;
    }
    virtual ~BaseException() { if (str) delete [] str; }
    const char *desc() const
    { return str; }
};
```

```
void BaseException::copy(const char *s)
{
    if (str)
         delete [] str;
    str = new char[strlen(s) + 1];
    strcpy(str, s);
}
```

המחלקה BaseException מייצגת את מצב השגיאה הבסיסי, כשהמצביע str מצביע למחרוזת המתארת את הבעיה. מכיון שיש למחלקה מצביע לתווים שמוקצה בעזרת האופרטור new, **יש להגדיר בנאי העתקה ואופרטור השמה**. אם לא נבצע הגדרות אלו, נימָצא במצב בו יש מצביעים משותפים בין אובייקטים לאותו אזור זיכרון (עובדה שיכולה לגרום לבעיות רבות, כמו, שחרור כפול של אותו אזור זיכרון).

המחלקה SafeArray יורשת מההמחלקה BaseException (שגיאה בסיסית). בנאי השגיאה RangeError מעביר מחרוזת קבועה לבנאי השגיאה הבסיסית BaseException. אופרטור האינדקס [] מעורר שגיאה מסוג RangeError כשיש פנייה לאינדקס מחוץ לתחום של המערך.

```
template <class T>
class SafeArray {
     T *arr;
     int size;
public:
     struct RangeError : public BaseException {
          int size;
          int bad_index;
          RangeError(int sz, int idx)
          : BaseException("SafeArray range error"),
            size(sz), bad_index(idx)
          { }
     };
     SafeArray(int sz) { arr = new T[size=sz]; }
     T &operator[](int i)
     { if (i>=size || i<0)
          throw RangeError(size, i);
       return arr[i];
     }
};

void fill_array(SafeArray<int> &sa)
{
     for (int i=0; i<20; i++)
          sa[i] = i;
}
```

הפונקציה הראשית, main, לוכדת כעת שני מצבי שגיאה. מצב השגיאה הראשון הוא המצב המסוים יותר, מצב השגיאה היורש. מצב השגיאה הבסיסי תופס את השגיאה BaseException. אם היינו משנים את סדר פונקציות הטיפול, הרי שלעולם לא היתה מתבצעת שגרת הטיפול עבור RangeError, מכיון שכל אובייקט מסוג RangeError הוא גם אובייקט מסוג BaseException. לכן, שגרת הטיפול הראשונה מתאימה גם לשגיאות טווח. מאחר והמערכת בודקת פונקציות טיפול מהראשונה והלאה, הרי ששגרת הטיפול הראשונה תימצא מתאימה ותופעל.

```
int main()
{
    SafeArray<int> sa(10);
    try {
         fill_array(sa);
    }
    catch (SafeArray<int>::RangeError re) {
         cout << "caught range error size:" << re.size
              << ", bad_index:" << re.bad_index << endl;
    }

    catch (BaseException &be) {
         cout << be.desc() << endl;
    }

    return 0;
}
```

### 8.3.4.2 שימוש בפונקציות וירטואליות

אם היינו משנים את הפונקציה desc לפונקציה וירטואלית, היינו יכולים להשתמש במצבי שגיאה בצורה פולימורפית, בדרך הבאה:

```
class BaseExcpetion {
...
   virtual void desc(ostream &out) const
   { out << str; }
};

template <class T>
class SafeArray {
public:
   struct RangeError : public BaseException {
   //...
   void desc(ostream &out) const {
      out << BaseException::desc()
```

```
            << " size=" << size
            << ", bad_index=" << bad_index << endl;
        }
    };
    //...
};
```

כלומר, במקום להחזיר מצביע לתווים, מדפיסה הפונקציה desc את המחרוזת לפלט הסטנדרטי. בנוסף לזה, הפונקציה היא פונקציה וירטואלית, ולכן השימוש בה דרך מצביע או ייחוס לאובייקט, יפעיל את הפונקציה המתאימה לאובייקט.

לאחר שהשתמשנו בפונקציה וירטואלית המציגה את הבעיה למשתמש, נוכל לשנות את הפונקציה הראשית בדרך הבאה:

```
int main()
{
     SafeArray<int> sa(10);
     try {
          fill_array(sa);
     }
     catch (BaseException &be) {
          cout << be.desc() << endl;
     }

     return 0;
}
```

כעת מפעילה שגרת הטיפול את הפונקציה הווירטואלית desc דרך ייחוס. הפונקציה הווירטואלית של אובייקט מסוג RangeError תופעל באמצעות ייחוס, כי אובייקט השגיאה גם הוא מסוג זה.

כדי שנוכל לתקן את מצב השגיאה, צריכים לדעת את סוג השגיאה המסוים. הגישה הווירטואלית פועלת כשלא רוצים, או לא יכולים, להתגבר על השגיאה. השימוש בגישה הפולימורפית יעיל במיוחד לפונקציות ברמה גבוהה, שמפעילות פונקציות רבות אחרות, ולמעשה היא מטפלת רק בשגיאות מסוג זה.

### 8.3.4.3 לכידת כל השגיאות

ברמה גבוהה במיוחד, כמו הפונקציה הראשית למשל, יש היגיון ללכוד את כל השגיאות. ++C תומכת בלכידה של כל השגיאות. הדרך בה לוכדים את כל השגיאות מוצגת בשגרת טיפול הנראית כך:

```
int main(int argc, char **arrgv)
{
    try {
      // ...
```

```
    }
    catch (...) {
      // handle any exception...
    }
}
```

שגרת טיפול עם ארגומנט "..." תופסת את כל השגיאות האפשריות. במצב כזה יכולה הפונקציה לצאת מהתוכנית, תוך כדי הודעה מתאימה. מצב כזה עדיף על מצב בו התוכנית נכשלת ומפסיקה ללא כל מידע המתאר את הבעיה.

### 8.3.4.4 לעורר מחדש את השגיאה

C++ תומכת בעירור מחדש שגיאה. מעוררים מחדש מצב שגיאה במקרה שאובייקט הלוכד מצב שגיאה, מתקן את מצבו הפנימי. אבל האובייקט אינו יכול לתקן את מצב השגיאה עצמו, לכן הוא מעביר את מצב השגיאה לפונקציה הנמצאת מעליו בשרשרת הקריאות.

עירור מצב שגיאה מחדש - פעולה זו מושגת על ידי הפעלה של האופרטור **throw** ללא כל פרמטרים. כך נעשה זאת:

```
void f()
{
    try {
      // ... call some other functions here
    }
    catch (...) {
      // ... restore state
      throw;      // rethrow caught exception
    }
}
```

הפונקציה f תופסת כל מצב חריג המתעורר כתוצאה מפעילות של פונקציות אחרות שנקראות בתוך הבלוק של try. כשמזוהה מצב שכזה, מחזיר "בלוק הטיפול במצב החריג" את המצב לקדמותו, ואז, בעזרת הפעלה של האופרטור throw ללא כל ארגומנטים, מעורר את המצב החריג שנתפס שוב.

### 8.3.4.5 סימון מצבים חריגים

פונקציה יכולה לסמן את המצבים החריגים שהיא מעוררת. הדבר מסייע לפונקציות הקוראות לפונקציה זו לזהות את סוגי המצבים החריגים שמעוררת הפונקציה הנקראת, ולאפשר לפונקציה הקוראת להיערך לקראת מצבים כגון אלה.

סימון המצבים החריגים נעשה במקום ההצהרה של הפונקציה, באופן הבא:

```
struct X1 {...};
struct X2 {...};
```

```
void g() throw (X1);
void f() throw(X1, X2);
```

הפונקציה f מצהירה, שרק מצבים חריגים מסוימים יכולים להתעורר במהלך הפעלתה. בדוגמה הקודמת מכריזה הפונקציה שרק מצבי X1 או X2 יכולים להתעורר כתוצאה מהפעלתה של f.

```
void g() throw(X1)
{
    // ... do some work
    throw X1;
}

void f() throw (X1, X2)
{
    g();
    //... some other work
    throw X2;
}
```

הפונקציה f קוראת לפונקציה g שמעוררת מצב חריג X1, ולכן היא מגדירה גם מצב חריג זה. אם הפונקציה f לא תגדיר את X1 כאחד מהמצבים החריגים שהיא מעוררת (אפילו אם לא באופן ישיר), תסתיים התוכנית בצורה בלתי צפויה. למשל:

```
struct EX1 {  };
struct EX2 {  };
struct EX3 {  };

void g() throw(EX2);

void f() throw (EX1, EX3)
{
   cout << "in f before g" << endl;
   g();
   cout << "in f after g" << endl;
   throw EX1();
}

void g() throw (EX2)
{
   cout << "in g" << endl;
   throw EX2();
}

int main()
{
```

```
    try {
         f();
    }
    catch (EX1 x1) {
         cout << "got X1" << endl;
    }
    catch (...) {
         cout << "got ... exeception" << endl;
    }
    return 0;
}
```

הדוגמה מציגה מקרה בו פונקציה מגדירה את EX1 ו-EX3 כמצבים חריגים שהיא מעוררת באופן ישיר ובלתי ישיר. הפונקציה f קוראת לפונקציה g שמעוררת מצב חריג מסוג EX2. מכאן נובע, שהפונקציה f מעוררת מצב חריג EX2 בצורה בלתי ישירה, בניגוד להכרזתה. בעת ההפעלה של הפונקציה f תיקרא פונקציית המערכת **unexpected** שתסיים את התוכנית. המשפט האחרון נכון, למרות שהפונקציה הראשית main לוכדת את כל המצבים החריגים במשפט catch האחרון.

כשהפונקציה אינה מגדירה רשימת חריגים שהיא עשויה לעורר, היא עלולה לעורר **כל** מצב חריג. לעומת זאת, אם הפונקציה מגדירה **רשימה ריקה** של חריגים, היא מתחייבת שלא לעורר כל חריג.

```
void f() throw();        // do not throw
void g();                // can throw any exception
```

בדוגמה זו f אינה מעוררת מצב חריג כלשהו, ואילו הפונקציה g יכולה לעורר כל מצב חריג. מומלץ להגדיר לכל פונקציה את המצבים החריגים שהיא עלולה לעורר.

### 8.3.4.6 טיפול בחריגים שאינם צפויים

ראינו בסעיף קודם, שיש מצבים בהם נקראת פונקציית המערכת **unexpected**. במקרים רבים לא מקובל לסיים תוכנית בצורה זו, ולכן אפשר לגרום לפונקציית המערכת **unexpected** לקרוא לפונקציה אחרת של המשתמש, במקום לסיים את התוכנית. דבר זה ניתן לביצוע בעזרת פונקציית המערכת **set_unexpected**.

```
typedef void (*PFV)();
   // declare a pointer to a void returning function
PFV set_unexpected(PFV);
```

הפונקציה set_unexpected מקבלת מצביע לפונקציה לה היא קוראת, במידה והפונקציה unexpected נקראת, כלומר קרה מצב חריג שלא נלכד, או פונקציה עוררה מצב חריג שלא הוגדר על ידה. הפונקציה set_unexpected מחזירה מצביע לפונקציה הקודמת שהיתה רשומה למערכת. מצב כזה מאפשר ליצור מחלקה המטפלת ברישום של פונקציות כאלו:

```
class set_unexpected_func {
    PFV func;
public:
    set_unexpected_func(PFV f)
    { func = set_unexpected(f); }
    ~set_unexpected_func()
    { set_unexpected(func); }
};
```

מחלקה זו מאפשרת לרשום פונקציה המטפלת במקרים חריגים שאינם מטופלים. בבנאי של המחלקה נרשמת שגרת טיפול למצבים כאלה. במפרק של המחלקה מבוטלת הפונקציה הזו והמצב חוזר לקדמותו. כלומר, כשאובייקט של מחלקה זו מוגדר, הוא רושם שגרת טיפול במצבים בלתי צפויים, וכשהוא יוצא מתחום ההגדרה, מבוטלת הפונקציה שנרשמה בבנאי. כעת אנו יכולים להשתמש באובייקטים כאלה בדרך הבאה:

```
void undefined_exception()
{
    cout << "undefined exception has been thrwon" << endl;
    exit(0);
}

void f()
{
    //...
    g();
}

int main()
{
    set_unexpected_func suf(undefined_exception);
    f();
    return 1;
}
```

## 8.4 שימוש וטכניקות טיפול במצבים חריגים

עד עתה ראינו את מכניזם הטיפול במצבים חריגים. לאחר שלמדנו את התחביר והמינוח של טיפול בחריגים, הגענו למצב בו ניתן ללמוד טכניקות שונות לטיפול במצבים חריגים. נְפַתֵּח גם מספר כלים לטיפול במצבים אלה.

מצבים חריגים המתעוררים בעזרת האופרטור **throw** נראים, במבט ראשון, כדבר קל לשימוש, אולם לא כך הדבר. שימוש לא מבוקר במצבים חריגים עלול לגרום לתקלות חמורות. לכן, יש להשתמש באמצעים אלה (עירור מצבים חריגים) בתשומת לב רבה.

**זכור**, קל מאוד לעורר מצבים חריגים, אך קשה ללכוד אותם ולטפל בהם כיאות!

## 8.4.1 טיפול בזיכרון

כשמקצים זיכרון בעזרת קריאה לאופרטור new, ומתעורר מצב חריג במהלך קריאה לפונקציה, לא ישוחרר הזיכרון שהוקצה, באופן אוטומטי. למשל:

```
void g() throw (X1);

void f() throw (X1)
{
    char *ptr = new char[5];
    // ... some work
    g();
    delete [] ptr;
}
```

במצב שתואר קודם לכן מקצה הפונקציה f זיכרון לצורך עבודתה, ולאחר מכן קוראת לפונקציה g; לסיום, הפונקציה f משחררת את הזיכרון שהקצתה. נראה פשוט? אבל מצב העניינים אינו כזה, כפי שנראה מייד.

אם g אכן מעוררת מצב חריג, אז הבקרה של התוכנית כלל אינה חוזרת לפונקציה f, ולפיכך הזיכרון שהוקצה ב-f אינו משוחרר. כלומר, לפנינו **זליגת זיכרון** (memory leak, קטע זיכרון שאובד למערכת מכיון שאין משחררים אותו). תוכניות שאינן משחררות זיכרון נראות תקינות בתחילה, אבל לאחר זמן מה מפסיקה התוכנית לתפקד באופן בלתי צפוי.

לו היינו מבצעים את הקצאת הזיכרון באובייקט ומשחררים את הזיכרון במפרק של האובייקט, היינו נמנעים מבעיה זו. הסיבה לכך היא, שמצב חריג שמתעורר גורם לניקוי מחסנית (stack) התוכנית, וכל מפרקי האובייקטים שמוגדרים במחסנית מבוצעים.

```
class ptr {
    char *p;
public:
    ptr(char *pc) : p(pc) {}
    ~ptr() { delete [] ptr; }
    operator char*() { return p; }
};

void f() throw (X1)
{
    ptr p(new char[5]);
    // ... some work
    g();
}
```

במצב המתואר אין צורך לשחרר את הזיכרון שמוקצה בפונקציה. מפרק המחלקה ptr משחרר את הזיכרון ונקרא כשהפונקציה מסיימת את עבודה בצורה רגילה, או כתוצאה ממצב חריג.

כדי לטפל בבעיות מסוג זה נפתֵח שני סוגי מחלקות. המחלקה הראשונה תטפל בהקצאת אובייקטים והשנייה תטפל בהקצאת מערכי אובייקטים. בסעיפים הבאים נראה שתי מחלקות אלו.

## 8.4.1.1 מצביעים אוטומטיים לאובייקטים

**מצביעים אוטומטיים** (auto pointers) הם מצביעים לאובייקטים שהוקצו מהזיכרון הדינמי של התוכנית, בעזרת האופרטור new. מצביעים אוטומטיים משחררים את המצביע שאליו מתייחסים במפרקים שלהם. פרט לכך, מצביעים אלה מתנהגים כמצביעים רגילים.

המחלקה **auto_ptr** מייצגת את המצביע האוטומטי. מצביע כזה נקרא, בסלנג של מתכנתי ++C, **מצביע חכם** (smart pointer). מחלקה זו היא מחלקת תבנית שהפרמטר שלה הוא סוג האובייקט שאליו היא מצביעה.

אובייקט מסוג auto_ptr מאותחל בבנאי שמקבל מצביע מסוג מתאים. במקרה אחר, האתחול הוא בעזרת בנאי העתקה. כשמאותחל אובייקט מסוג auto_ptr בעזרת אובייקט אחר מאותו סוג, יש לשחרר את המצביע של האובייקט השני, כדי למנוע שחרור כפול של המצביע שאליו מתייחסים שני האובייקטים.

שחרור המצביע מתבצע על ידי הפונקציה release, שמאפסת את המצביע ומחזירה מצביע לערך הקודם שלו. לפי הגדרת השפה, הפעלת אופרטור delete על זיכרון מאופס אינה מבצעת דבר, ולכן איפוס המצביע הכרחי ומספיק.

הפונקציה swap מחליפה בין המצביע הנוכחי למצביע הנתון לפונקציה כארגומנט. הפונקציה remove משחררת את הזיכרון שאליו מצביע המצביע של האובייקט הנוכחי.

```
//----------------------------------------------------------
// auto_ptr - is a smart pointer which enables its user to
// delete the pointed pointer automatically when it goes
// out of scope
//----------------------------------------------------------
template <class T>
class auto_ptr {
     T *ptr;         // the pointer
public:
     auto_ptr(T *p=0) : ptr(p) {}
     auto_ptr(auto_ptr<T> &ap) : ptr(ap.release())
     {}
     ~auto_ptr() { delete ptr; }
     auto_ptr<T> &operator=(auto_ptr<T> &rhs)
```

```
     { reset(rhs.ptr); return *this; }
     T &operator*() const { return *ptr; }
     T *operator->() const { return ptr; }
     T *get() const { return ptr; }
     T *release() { T *p=ptr; ptr=0; return p; }
     void reset(T *p=0);
     T *swap(T *p)
     { T *tp = ptr; ptr = p; return tp; }
     static void remove(T *&p)
     { T *tp=p; p=0; delete tp; }
};
```

הפונקציה המסובכת ביותר בנושא זה היא זו שמבצעת השמה מחדש למצביע. הפונקציה reset מקבלת מצביע, ומשחררת את המצביע הקודם, אם אינו זהה למצביע הנתון לה. לאחר מכן, משימה הפונקציה את המצביע הנתון למצביע של האובייקט הנוכחי. כלומר, הפונקציה מחליפה את המצביע של האובייקט הנוכחי במצביע נתון.

```
// reset the old pointer to the given one. If old pointer
// is not the same as given one deletes it
template <class T>
inline void auto_ptr<T>::reset(T *p)
{
     T *tp = ptr;
     ptr = p;
     if (tp != p) delete tp;
}
```

השימוש במחלקה זו הוא כזה:

```
class auto_test {
     int val;
public:
   auto_test(int v=0) { val=v; }
   ~auto_test()
   { cout << "auto_test::~auto_test val=" << val << endl;
}
   auto_test &operator=(int v)
   { val = v; return *this; }
};
```

המחלקה auto_test הינה דוגמה לאובייקטים שעליהם יכול **המצביע החכם** להצביע. ההדפסות במפרק של מחלקה זו נועדו להדגים את פעולת המצביע החכם ולהציג פירוק אובייקט.

```
struct EX1 {  };
struct EX2 {  };
struct EX3 {  };
```

```
void g() throw(EX2);

void f() throw (EX1, EX3)
{
     auto_ptr<auto_test> p = new auto_test;
     *p = 1;
     cout << "in f before g" << endl;
     g();
     cout << "in f after g" << endl;
     throw EX1();
}
```

הפונקציה f מקצה אובייקט מסוג auto_test, ולאחר מכן מעדכנת את ערכו. מכיון שהפונקציה g מעוררת מצב חריג, אין התוכנית מגיעה לשורת ההדפסה השנייה, ובכל זאת נקרא המפרק של האובייקט auto_test. פלט ההדפסה של קטע הקוד הוא :

```
auto_test::~auto_test val=1
```

## 8.4.1.2 מצביע אוטומטי למערך של אובייקטים

**מצביע אוטומטי למערך של אובייקטים** (auto array pointer) מוקצה בעזרת האופרטור []new, אופרטור ההקצאה של מערך. מחלקה זו היא מחלקת תבנית ופרמטר התבנית הוא סוג המערך. במחלקה זו יש פונקציות דומות לאלט שבמחלקה הקודמת, פרט לעובדה שיש התייחסות לכך שהמצביע הוא מצביע למערך.

האופרטור הנוסף במחלקה auto_ptr הוא אופרטור הפנייה לאינדקס ([]) של המערך. אופרטור זה מחזיר ייחוס לאיבר נתון במערך. דבר זה מקנה למחלקה התנהגות כמו מצביע למערך רגיל של השפה.

כשמפעילים את אופרטור delete עבור המצביע ptr, משתמשים באופרטור שחרור מערכים. גם כאן וגם במחלקה הקודמת, האחריות על מתן סוג המצביע הנכון מוטלת על המשתמש. כלומר, אם ניתן מצביע מסוג T, המצביע למערך במחלקה הקודמת, או מצביע המצביע לאובייקט מסוג T במחלקה הנוכחית, התוצאות אינן מוגדרות.

```
//----------------------------------------------------------
// auto_array_ptr is a smart pointer to a builtin array.
// The smart pointer deletes the underlined array when it
// goes out of scope.
//----------------------------------------------------------
template <class T>
class auto_array_ptr {
    T *ptr;
public:
    auto_array_ptr(T *p=0) : ptr(p) {}
```

```
    auto_array_ptr(auto_array_ptr<T> &ap) :
            ptr(ap.release())
    {}
    ~auto_array_ptr() { delete [] ptr; }
    auto_array_ptr<T> &operator=(auto_array_ptr<T> &rhs)
    { reset(rhs.ptr); return *this; }
    T &operator*() const { return *ptr; }
    T *operator->() const { return ptr; }
    T &operator[](int i)  { return ptr[i]; }
    T *get() const { return ptr; }
    T *release() { T *p=ptr; ptr=0; return p; }
    void reset(T *p=0);
    T *swap(T *p)
    { T *tp = ptr; ptr = p; return tp; }
    static void remove(T *&p)
    { T *tp=p; p=0; delete [] tp; }
};

// resets the pointer to the array to the given pointer. If
// the given pointer is not the same as old pointer then
// deletes old pointer
template <class T>
void auto_array_ptr<T>::reset(T *p)
{
    T *tp = ptr;
    ptr = p;
    if (tp != p)
            delete [] tp;
}
```

אופן השימוש במחלקה זו פשוט:

```
void f() throw (EX1, EX3)
{
    const int arrsize = 5;
    auto_array_ptr<auto_test> p = new auto_test[arrsize];
    for (int i=0; i<arrsize; i++)
            p[i] = i;
    cout << "in f before g" << endl;
    g();
    cout << "in f after g" << endl;
    throw EX1();
}
```

השימוש באובייקט p הוא כמו שימוש רגיל במצביע למערך אובייקטים בשפה. הלולאה בפונקציה מכניסה ערכים עוקבים, החל באפס, אל האיברים העוקבים של המערך. הפונקציה g מעוררת מצב חריג, ולכן לא תבוצע שורת ההדפסה האחרונה בפונקציה f. לעומת זאת, המפרקים של האובייקטים הנמצאים במערך ייקָראו, ופלט קטע תוכנית זו נקרא כך:

```
in f before g
in g
auto_test::~auto_test val=4
auto_test::~auto_test val=3
auto_test::~auto_test val=2
auto_test::~auto_test val=1
auto_test::~auto_test val=0
```

## 8.4.2 שימוש במשאבים

זיכרון המחשב הוא אחד ממשאבי המחשב. משאבים אחרים של המחשב יכולים להיות **מתארי קבצים** (file descriptors), אשר לעתים מכנים אותם **מצביעים לקבצים**, או זיכרון משותף למספר תהליכים. כאשר נרכשים משאבים כאלה יש צורך לשחרר אותם, אחרת התוכנית אינה יכולה לתפקד באופן תקין. בסעיף זה נלמד טכניקה העוזרת להתגבר על מצבים כאלה.

הטכניקה נקראת באנגלית: resource acquisition is initialisation. משמעותה בעברית **רכישת המשאב בזמן אתחולו** (או לכידת המשאב היא אתחול).

כבר הפעלנו טכניקה זו להקצאות זיכרון כשהגדרנו את שתי המחלקות (auto_ptr). העיקרון שהנחה אותנו היה שהמפרקים של האובייקטים שנמצאים במחסנית של התוכנית נקראים גם אם מתעורר מצב חריג. לפיכך, עטפנו את הקצאת הזיכרון באובייקט ששחרר את הזיכרון בעת הצורך.

נניח שיש לנו פונקציה שקוראת קובץ לחוצץ. הפונקציה מקצה עבור החוצץ קטע זיכרון המתאים לגודל הקובץ, והיא עושה זאת כך:

```
char *read_file(consr char *name)
{
    stat st;
    stat(&st);
    int file_size = st.st_size, fd;
    fd = open(name, RD_ONLY);
    char *buf = new char[file_size];
    read(fd, buf, file_size);
    close(fd);  // return the resource to the system
    return buf;
}
```

הפונקציה משתמשת בפונקציית המערכת stat המחזירה מידע על הקובץ. בין היתר, מוחזר מידע על גודל הקובץ. הפונקציה open פותחת את הקובץ לקריאה ומחזירה ידית לקובץ. בידית זו יש להשתמש כשמשתמשים בפונקציה read. לשם פשטות, בדיקת השגיאות אינה נעשית כאן.

בקטע הקוד הקודם עלולה להתרחש בעיית הקצאת זיכרון. כלומר, אם הקובץ גדול אין מספיק זיכרון. כשאין מספיק זיכרון יכולה המערכת לעורר מצב חריג bad_alloc. לאופרטור new יש אפשרות לקרוא לפונקציה המטפלת במקרה זה, אם הפונקציה מועברת למערכת דרך הפונקציה **set_new_handler**. לדוגמה:

```
void out_of_memory()
{
    cerr << "new failed to allocate memory" << endl;
    exit(1);
}
//...
set_new_handler(out_of_memory);
```

נניח שדבר זה לא נעשה, ושהמערכת אכן מעוררת מצב חריג. כתוצאה מכך, אין השורה הסוגרת את הקובץ מופעלת, ולכן יש זליגה של ידיות קבצים. הפתרון המתבקש הוא לעטוף את פתיחת הקובץ באובייקט שסוגר את הקובץ כשהאובייקט יוצא מתחום השליטה שלו. כלומר:

```
class file {
    int fd;
public:
    file(const char *nm) { fd = open(nm, RD_ONLY); };
    ~file() { close(fd); }
    operator int() { return fd; }
};
```

כעת יכולה פונקציה כמו read_file, שהשתמשנו בה קודם, להיכתב כך:

```
char *read_file(consr char *name)
{
    stat st;
    stat(&st);
    int file_size = st.st_size;
    file fd(name);
    char *buf = new char[file_size];
    read(fd, buf, file_size);
    return buf;
}
```

בכל מצב חריג ייקרא המפרק של האובייקט fd, ולכן תשוחרר ידית הקובץ למערכת.

אילו היינו משתמשים באובייקט מסוג ifstream לא היינו צריכים להוסיף מחלקה נוספת, שכן אובייקט זה סוגר את הקובץ המחובר אליו, במפרק שלו. אבל הדוגמה לעיל נעשתה כדי להמחיש משאבים שיכולים להיות שונים מקבצים.

## 8.5 שילוב בין תבניות למצבים חריגים

כשמשתמשים בתבניות (מחלקות או פונקציות) איננו יודעים הרבה על האובייקטים של פרמטר התבנית. כתוצאה מכך, איננו יודעים גם את סוג החריגים שמעוררים אובייקטים אלה. למשל:

```
template <class T>
class Array {
    T *arr;
    int size;
public:
    Array(int sz) : arr(0), size(0) { alloc(sz); }
    void alloc(int sz)
    { T *tmp = new T[sz];
      //...
    }
};
```

כשמקצים מערך של אובייקטים מסוג T, נקראים הבנאים חסרי הארגומנטים (בנאי ברירת המחדל) של האובייקטים מסוג T. בנאי זה יכול לעורר מצב חריג. כמפתחי המחלקה Array, איננו יודעים אילו מצבים חריגים מסוגל אובייקט מסוג T לעורר. מצב זה קיים גם בפונקציות פשוטות כמו:

```
template <class T>
void swap(T &v1, T &v2)
{
     T tmp(v1);
     v1 = v2;
     v2 = tmp;
}
```

כל אחת משלוש השורות של פונקציית התבנית swap יכולה לעורר מצב חריג. השורה הראשונה משתמשת בבנאי ההעתקה של אובייקט מסוג T. אם בנאי זה מנסה לרכוש משאב ולא מצליח, עלול הבנאי לעורר מצב חריג. שתי השורות האחרונות משתמשות באופרטור ההשמה, שיכול לעורר מצב חריג מסיבות הדומות לאלו של בנאי ההעתקה.

דוגמאות אלו מסבירות גם את הסיבה לכך שפונקציה, שלא מצהירה על המצבים החריגים שהיא מעוררת, יכולה לעורר כל מצב. פונקציית תבנית אינה יכולה להצהיר על סוגי מצבים חריגים המתעוררים כתוצאה מהפעלתה!

בסעיף זה ניקח דוגמה של מחלקת תבנית המייצגת מחסנית, ונראה כיצד מזהים את המצבים החריגים ואיך מטפלים בהם.

## 8.5.1 מחלקת התבנית - מחסנית

**מחסנית** (stack) היא מבנה נתונים שבו האובייקט האחרון שנכנס הוא הראשון שיוצא. כלומר, המחסנית מקיימת את החוק הבסיסי: נכנס אחרון יוצא ראשון.

המחסנית שנממש כאן בנויה על מערך אובייקטים (array) כש-top מציין מיקום אחד מעבר לאובייקט העליון במחסנית. השדה end_of_storage הוא אינדקס אחד מעבר לסוף המחסנית.

כאשר המחסנית מלאה, היא מקצה מקום חדש, שגודלו **כפול** מזה שהיה ועוד אלמנט אחד בסוף. כאשר גודל המחסנית הוא אפס, היא מקצה מקום **אחד** בלבד! מדיניות ההקצאה של המחסנית גורמת לה לגדול במהירות. הדבר דומה לאלגוריתם של המחלקה Array, שראינו בפרקים הקודמים בספר זה. כאן מוקצה מערך של אובייקטים, ולכך שני חסרונות עיקריים:

- לאובייקטים המוכנסים למחסנית חייב להיות בנאי ברירת המחדל.
- הקצאת מערך של אובייקטים גורמת להפעלת בנאי המערך על כל המערך. מכאן, שייתכן שבנאים יופעלו (שקול לקריאה לפונקציה) עבור אובייקטים שלא השתמשו בהם.

למרות חסרונות אלה נרצה לראות את בעיות השילוב בין תבניות למצבים חריגים, ולכן לא נסבך את הדוגמה, אלא נפתור בעיות אלו.

הפונקציה push דוחפת אובייקט נוסף למחסנית, והפונקציה pop מוציאה אובייקט מהמחסנית. אם המחסנית ריקה, מעוררת פונקציה זו מצב חריג. הפונקציה is_empty מחזירה ערך השונה מאפס, אם המחסנית ריקה.

```
template <class T>
class Stack {
    T *array;
    int top;                // last element index
    int end_of_storage;     // capacity of array
    void allocate(int size);
public:
    Stack() : array(0), top(0), end_of_storage(0)
    {}
    ~Stack() { delete [] array; }
    T pop() {
            if (top <= 0)
                    throw "Empty stack";
            return array[--top];    // <ex1>
    }
    void push(const T &val) {
            if (top >= end_of_storage)
                    allocate(end_of_storage * 2+1);
            array[top++] = val;     // <ex2>
```

```
    }
    int is_empty() const { return (top <= 0); }
};
```

הפונקציה allocate מקצה זיכרון בגודל נתון. לאחר ההקצאה מועתקת תכולתו הקודמת של המערך לאזור החדש ואז משוחרר האזור הישן.

```
template <class T>
void Stack<T>::allocate(int sz)
{
    T *tmparr = new T[sz];          // <ex3>
    for (int i=0; i<end_of_storage; i++)
            tmparr[i] = array[i];   // <ex4>
    delete [] array;
    array = tmparr;
    end_of_storage = sz;
}
```

## 8.5.2 השימוש במחלקה Stack

השימוש במחלקה זו פשוט למדי. אובייקטים מוכנסים למחלקה בעזרת הפונקציה push ומוצאים ממנה על ידי הפונקציה pop. נגדיר מחלקה Text המייצגת מחרוזות ונכניס אותה למחסנית. המחלקה Text מחזיקה מצביע לאזור בזיכרון שאליו מועתקת המחרוזת הנתונה לבנאי של מחלקה זו.

כשיש מחלקה ולה מצביעים למחרוזות, מומלץ להגדיר למחלקה בנאי העתקה ואופרטור השמה, כדי למנוע הצבעה כפולה לאותו אזור זיכרון. כתוצאה מהצבעה כזו עלול אזור הזיכרון להיות משוחרר פעמיים. המחלקה משתמשת בפונקציות המערכת המקצות זיכרון (strdup) והמשחררות אותו (free).

```
class Text {
    char *str;
    void copy(const char *s)
    { if (str) free(str); str = strdup(s); }
public:
    Text(const char *s="")  { str=0; copy(s); }
    ~Text()                 { free(str); }
    Text(const Text &t)     { str=0; copy(t.str); }
    Text &operator=(const Text &t)
                            { copy(t.str); return *this; }
    operator const char *() { return str; }
    operator int()          { return atoi(str); }
    operator float()        { return atof(str); }
};
```

הפונקציה push מכניסה למחסנית max אובייקטים מסוג Text. הפונקציה מקבלת ייחוס למחסנית של אובייקטים מסוג Text.

```
void push(Stack<Text> &s, int max)
{
    char buf[32];
    for (int i=0; i<max; i++) {
         ostrstream ostr(buf, 32);
         ostr << i << ends;
         Text t(buf);
         s.push(t);
    }
}
```

הפונקציה pop מוציאה אובייקטים מהמחסנית. הפונקציה מוציאה אובייקטים כל עוד המחסנית אינה ריקה. האובייקטים שהוצאו מודפסים לפלט הסטנדרטי.

```
void pop(Stack<Text> &s)
{
    while (!s.is_empty()) {
         cout << int(s.pop()) << " ";
    }
    cout << endl;
}
```

הפונקציה הראשית מגדירה מחסנית וקוראת לשתי הפונקציות הראשונות כדי להכניס, להוציא ולהדפיס את האובייקטים שבמחסנית.

```
int main()
{
    Stack<Text> s;
    push(s, 25);
    pop(s);
    return 1;
}
```

האובייקטים מודפסים בסדר הפוך להכנסתם, מהגדול לקטן.

פלט התוכנית ייראה כך:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

## 8.5.3 טיפול במצבים חריגים

בקטעי הקוד שראינו בסעיפים קודמים, היו מספר מצבים חריגים שהטיפול בהם לוקה בחסר. בסעיף זה נלמד כיצד לטפל במצבים אלה במקומות שאפשר. ייתכנו מקומות בהם אין פתרון טוב יותר. מקומות אלה נתאר בהמשך.

ניתוח של מצבים חריגים יכול להתבצע לאורך שני מסלולים:

- **מצבים חריגים מקומיים**. מצבים חריגים שמתעוררים על ידי מחלקת התבנית.
- **מצבים חריגים חיצוניים.** מצבים חריגים שמעוררים האובייקטים שמשמשים כפרמטר של התבנית.

הסוג הראשון של מצבים חריגים קל לזיהוי משום שאנו, כמפתחי המחלקה, יודעים מהם המצבים שמעוררת מחלקת התבנית. לעומת זאת, המצבים החריגים שמתעוררים על ידי פרמטר של התבנית אינם ידועים לנו, לכן הטיפול בהם קשה יותר.

## 8.5.3.1 מצבים חריגים מקומיים

במחלקה זו יש מצב חריג מקומי יחיד שמעוררת אותו הפונקציה pop. לא השתמשנו בעובדה זו. רצוי לציין את סוגי המצבים החריגים שמעוררת פונקציה. במקרה שלא יודעים מהם החריגים שפרמטר התבנית מעורר, אין אפשרות לסמן את המצבים החריגים שהפונקציה מעוררת, ולכן השארנו את הפונקציה ללא סימון כזה.

במקרה זה, אין בעיה במחלקה Stack עצמה, מכיון שלא מתבצע דבר כשהמחסנית ריקה. במצב זה מעוררת הפונקציה pop מצב חריג ובזה היא מסתיימת. בקטע הקוד המשתמש במחסנית בדקנו לפני כל קריאה לפונקציה pop, אם המחסנית אינה ריקה ורק אז קראנו לה.

## 8.5.3.2 מצבים חריגים חיצוניים

מצבים חריגים חיצוניים מתעוררים על ידי פרמטר התבנית של המחלקה. המקומות האפשריים למצבים חריגים כאלה מסומנים בהערות וממוספרים בתוספת הקידומת ex. נעבור עתה על מקומות אלה וננתח מצב אחר מצב.

#### הפונקציה pop

פונקציה זו מחזירה העתק של האובייקט האחרון שנמצא במחסנית, אשר ממומש על ידי קריאה לבנאי ההעתקה של האובייקט, או קריאה לאופרטור ההשמה. אם אחד מהם מעורר מצב חריג, זה יקרה **לאחר** החזרת הפונקציה, ואין שום אפשרות לדעת ולגלות זאת בפונקציה pop עצמה.

```
T pop() {
     if (top <= 0)
          throw "Empty stack";
     return array[--top];     // <ex1>
}
```

הדבר היחיד שניתן לעשות, הוא לאפשר הגדלה של top על ידי המשתמשים במחלקה. אם נוקטים בשיטה זו, מוטל על המשתמש במחלקה לגלות את הבעיה ולטפל בה בעצמו. לכן, גישה זו אינה מומלצת.

הגישה השנייה, היא להתריע בפני המשתמש על אפשרות של מצב חריג, כך שהוא ידע שקיים חשש שהאובייקט בראש המחסנית יאבד ואז לא תהיה לו גישה אל האובייקט. לצערנו, אין פתרון מספק לבעיה זו. המלצתי האישית היא להימנע במקרים כאלה משימוש במצבים חריגים, ולהשתמש בהם רק במקומות שזה הכרחי.

כאמור, קל לעורר מצבים חריגים, אך קשה מאוד לטפל בהם!

### הפונקציה push

הפונקציה push קוראת לפונקציה allocate, אשר נדון בה מיד. כשמעתיקים את הערך הנתון לראש המחסנית, יכול אופרטור ההשמה של אובייקט מסוג T (הפרמטר של התבנית) לעורר מצב חריג. לכן, יש למנוע את הגדלתו של top, ולבצע את ההגדלה רק לאחר שההשמה אכן התבצעה. כך שומרים על שלמות מבנה הנתונים של המחסנית.

```
void push(const T &val) {
      if (top >= end_of_storage)
            allocate(end_of_storage * 2+1);
      array[top] = val;      // <ex2>
      top++;
}
```

### הפונקציה allocate

בפונקציה זו יש מספר רב של אפשרויות למצבים חריגים. במצב הראשון, כשמקצים מערך של אובייקטים מסוג T, נקרא **בנאי ברירת המחדל** (default condition) של אובייקטים אלה. כדי להגן על מצב זה ועל שאר המצבים החריגים, נשתמש באובייקט מסוג auto_array_ptr שיכיל את המצביע למערך החדש שמקצים.

בלולאת ההעתקה של המערך החדש לישן משתמשים באופרטור ההשמה, ולכן ייתכן שהאחרון יעורר מצב חריג. אם מתעורר מצב חריג כזה, אזי המערך החדש נהרס על ידי המפרק של tmparr. בעיה זו נפתרה אף היא.

בסוף הפונקציה אנו דואגים להחליף מצביעים בין המערך החדש לישן. לכן ייהרס המערך הישן על ידי המפרק של tmparr.

```
template <class T>
void Stack<T>::allocate(int sz)
{
    auto_array_ptr<T> tmparr = new T[sz];  // <ex3>
    for (int i=0; i<end_of_storage; i++)
            tmparr[i] = array[i];          // <ex4>

    array = tmparr.swap(array);
    end_of_storage = sz;
}
```

ראינו אם כן, שהשימוש בכלים שפיתחנו עוזר להתגבר על בעיית שחרור זיכרון או משאבים. ללא כלים אלה, היו הבעיות קשות יותר והקוד המטפל בהן היה מסובך.

## 8.6 סיכום

בפרק זה למדנו על המושג **מצבים חריגים ב-++C.** ראינו למה צריך מצבים חריגים. הצורך במצבים חריגים נובע מכך שיש פונקציות ב-++C, כגון הבנאי, שאינן מחזירות ערך המודיע על הצלחה, או כישלון.

לטיפול במצבים חריגים התוספו בשפה מילות המפתח הבאות : **try**, **catch**, **throw**. כשפונקציה מכניסה בלוק תחת הקוד try, היא מסמנת למהדר שהיא תטפל במספר מסוים של מצבים חריגים שיתעוררו מפעולות בבלוק זה. מצב חריג מטופל מופיע לאחר בלוק try ומסומן בבלוק של catch. בלוק כזה דומה לפונקציה עם ארגומנט אחד המסמן את המצב החריג בו הוא מטפל. יכולים להיות מספר בלוקים של טיפול כזה,זה אחר זה.

**כשמתעורר מצב חריג**, נקראים המפרקים של האובייקטים שנמצאים במחסנית. ניצלנו זאת כדי לפתח טכניקות טיפול מתאימות במצבים חריגים. טכניקת הטיפול הראשונה היתה "רכישת משאב בזמן אתחולו" כשביטול המשאב (שחרור) נעשה על ידי המפרק שלו.

פיתחנו כלים לטיפול בזיכרון המוקצה בפונקציות החשופות למצבים חריגים. כלים אלה כוללים מצביעים חכמים לאובייקטים, אשר מפרקים את האובייקטים הקשורים איתם ומשחררים את הזיכרון הקשור אליהם. לכן, כשמשתמשים במצביעים חכמים בפונקציות הצפויות למצבים חריגים, מובטח לנו שהזיכרון ישתחרר אם וכאשר מתעורר מצב חריג. בנוסף, מובטח לנו שהמפרקים שלהם ייקָראו ואז ישתחררו המשאבים שהוקצו על ידי האובייקטים.

ראינו כיצד יש לנתח קוד המשלב תבניות ומצבים חריגים. כשכותבים קוד הכולל בתוכו תבניות, איננו יודעים מה הם סוגי המצבים החריגים שיתעוררו, ולכן דבר זה קשה לפיתוח. ראינו, שקוד פשוט לכאורה, כמו מחסנית, צופן בחובו הרבה מצבים חריגים.

לסיכום, המלצתי היא : השתמש במצבים חריגים כשאין כל אפשרות אחרת. אם אפשר להחזיר ערך המסמן הצלחה או כישלון מפונקציה, השתמש בדרך זו ואל תעורר מצב חריג.

## 8.7 שאלות

1. האם קיימת דרך טובה יותר להתגבר על מצב חריג בלולאת ההעתקה בפונקציה pop של המחלקה מחסנית, כך שישמרו האובייקטים שהועתקו? האם יש יתרון לאפשרות כזו?
2. נתח את מחלקת התבנית Array. האם יש בה מקומות הצפויים למצבים חריגים? אם כן, כיצד יש לטפל בהם?
3. נתח את מחלקת התבנית list. האם יש במחלקה זו מקומות הצפויים למצבים חריגים? אם כן, טפל בהם.
4. האם אפשר להשתמש במחלקת תבנית כדי להגדיר מחלקה שתתגבר על בעיית שגיאה בבנאי, במקום להשתמש במצב חריג?
5. כתוב את המחלקה Stack בצורה יעילה יותר, כך שלא ייבנו האובייקטים כשמקצים מערך של אובייקטים. אובייקטים ייבנו רק כשמכניסים אובייקט למחסנית (רמז: השתמש באותה טכניקה כמו ב-Array).
6. נתח את המחלקה מהשאלה הקודמת וטפל במצבים חריגים שונים העלולים להתעורר.

## פרק 9

# חידושים ב-++C

בפרק זה נבחן מספר תוספות לשפת ++C שנודעו לאחרונה. **רוב המהדרים אינם תומכים עדיין בתוספות אלו** ולכן, הדוגמאות שמוצגות בפרק זה אינן רצות תחת רוב המהדרים. להרצת התוכניות בפרק זה השתמשנו **במהדרי בורלנד מגירסה 4 ומעלה.** אפשר להשתמש גם **במהדר מיקרוסופט** ובמהדרים אחרים בגירסה מקבילה ומעלה.

קבלת **מידע בזמן ריצה - RTTI** (**R**un **T**ime **T**ype **I**nformation) היא אפשרות שנוספה ל-++C לפני זמן לא רב. מטרתה לקבל מידע על אובייקטים בזמן שמריצים אותם. מידע בזמן ריצה מאפשר לנקוט בפעילות יעילה יותר כתלות בסוג האובייקט, גם אם הוא פוגם בתכונות הפולימורפיות של התוכנית. לפיכך, יש להשתמש באפשרויות אלו של השפה בתבונה.

**מרחב שמות** (name space) מאפשר שמירה על מרחב שמות נקי מהתנגשויות בין חלקי תוכנה בלתי תלויים (חשוב לזכור זאת, מכיון שבכל פרויקט גדול יש צורך בתיאום שמות). דבר זה מקל על מפתחי ספריות גנריות, ועל שימוש חוזר בחלקי תוכנה בלתי תלויים בפרויקטים גדולים.

המרות שונות של מצביעים, או של ייחוסים, מאפשרות לשנות את האובייקטים מקבועים לשאינם קבועים. נראה את סוגי המרות אלו בפרק זה. הן מבוצעות בעזרת מילות מפתח חדשות, ולכן קל יותר לזהות אותן.

## 9.1 מידע בזמן ריצה

שפת ++C תומכת במידע בזמן ריצה המאפשר קבלת מידע שונה, וברמות שונות בזמן ריצה. התמיכה במידע בזמן ריצה נתונה על ידי האופרטורים הבאים :

**typeid** - אופרטור המזהה את סוג האובייקט ומחזיר ייחוס לאובייקט מסוג **type_info**.

**type_info** - מחלקה בסיסית המוגדרת על ידי השפה ומסופקת עם המהדר.

**dynamic_cast** - התמרות (המרות מבנים) של מצביעים בזמן ריצה.

בסעיף זה נסקור את שלושת האופרטורים האלה.

## 9.1.1 האופרטור typeid

אופרטור זה מחזיר ייחוס לאובייקט מסוג **type_info** לאחר שהוא בודק אם הסוג של שני אובייקטים זהה. למשל, אם נגדיר את שתי המחלקות Base ו-Derive ואשר להן פונקציות וירטואליות, נוכל להשתמש באופרטור בצורה הבאה:

```
#include <iostream.h>
#include <typeinfo.h>

class Base {
public:
     Base() {}
     virtual ~Base() {}
     virtual const char *get_class_name() const
     { return "Base"; }
};

class Derive : public Base {
public:
     Derive() {}
     const char *get_class_name() const
     { return "Derive"; }
};

     Base b;
     Derive d;
     double dval;
     char cval;
     if (typeid(cval) == typeid(dval))
         cout << "oops something wrong" << endl;
     else
         cout << "typeid(cval) != typeid(dval)" << endl;
     if (typeid(b) == typeid(d))
         cout << "typeid(b) == typeid(d)" << endl;
     else
         cout << "typeid(b) != typeid(d)" << endl;
```

ניתן לבדוק סוגים של אובייקטים, או של משתנים בסיסיים בשפה, כגון double וכו'. האופרטור מאפשר לבדוק זהות, או שוני של סוגי משתנים. בשני משפטי if מתבצע ה-else ולכן הפלט שנקבל הוא:

```
typeid(cval) != typeid(dval)
typeid(b) != typeid(d)
```

סוג המשתנה cval וסוג המשתנה dval, שונים. גם סוג האובייקט b שונה מסוג האובייקט d.

גם אם היינו משתמשים במצביעים למחלקה הבסיסית שפונים לאובייקטים שונים, היינו מקבלים תוצאות מתאימות. לדוגמה:

```
Base *bp1, *bp2;
bp1 = &b;
bp2 = &d;
if (typeid(*bp1) == typeid(*bp2))
    cout << "*bp1 == *bp2 !!!" << endl;
else
    cout << "*bp1 != *bp2 !!!" << endl;
```

גם במקרה זה, למרות שאנו משתמשים במצביע מאותו סוג, הביטוי bp1* הוא אובייקט מסוג Base, והביטוי bp2* הוא אובייקט מסוג Derive. לכן, הפעלת האופרטור typeid על שני ביטויים אלה, יחזיר ערך אפס. במילים אחרות, אופרטור זה יודע להבדיל בין סוגי אובייקטים שונים, למרות שהתייחסות אליהם היא דרך מצביע למחלקה הבסיסית. לפיכך יהיה הפלט:

```
*bp1 != *bp2 !!!
```

האופרטור **typeid** מחזיר ייחוס לאובייקט מסוג **type_info**.

## 9.1.2 המחלקה type_info

המחלקה **type_info** מספקת מידע בסיסי על אובייקטים. אובייקט של מחלקה זו הוא התוצאה של האופרטור הבסיסי של השפה typeid. המחלקה type_info מוגדרת כך:

```
class type_info {
  // implementation-dependent
private:
  type_info(const typ_info&);   // no copy is allowed
  type_info &operator=(consty type_info &);
public:
  virtual ~type_info();         // polymorphic

  int operator==(const type_info &) const;  // can be compare
  int operator!=(const type_info &) const;
  int before(const type_id &) const;       // ordering

  const char *name() const;     // name of type
};
```

הדבר המעניין הראשון שניתן לראות הוא, שאין בנאים בחלק הציבורי של המחלקה האחרונה. לכן, אין למשתמש אפשרות להגדיר אובייקטים השייכים למחלקה זו. הבנאים בחלק הפרטי של המחלקה מונעים העתקה של אובייקטים מסוג type_info. אם לא היו בנאים אלה בחלק הפרטי היה המהדר מעניק בנאי העתקה ואופרטור העתקה אוטומטית לאובייקטים כאלה. גם אין אפשרות לגזור מחלקה אחרת מהמחלקה type_info, דבר שימושי כאשר רוצים להרחיב את הפונקציונליות של מחלקה זו.

האופרטורים "==" ו-"=!" מאפשרים לבדוק בצורה פולימורפית שוויון בין סוגים שונים של אובייקטים. בסעיף הקודם בדקנו שוויון בין סוגי אובייקטים, בעזרת האופרטור typeid. אופרטור זה מחזיר ייחוס לאובייקט מסוג type_info, ולכן המהדר מפעיל אופרטורים המתאימים של type_info, כדי לבדוק את השוויון.

הפונקציה before מאפשרת למיין אובייקטים מסוג type_info. הפונקציה מחזירה ערך אמת (1) אם האובייקט הנוכחי "קודם" לזה שנתון לה. הקדימות יחסית לסדר לקסיקוגרפי בין שמות המחלקות של האובייקטים ש-type_info מתייחס אליהם.

המאקרו הבא משתמש באופרטור typeid כדי להציג את יחס הסדר בין שני אובייקטים. אי אפשר להגדיר אובייקט מסוג type_info בצורה עצמאית, ולכן יש להשתמש באורפטור typeid. המאקרו משתמש בפונקציה before כדי להחליט על הסדר בין האובייקטים, וכדי לקבל את שמות המחלקה שלהם.

```
// display before display which one is before the other one
//
#define display_before(t1, t2)
\
{
\
    if (typeid(t1).before(typeid(t2))) {
\
          cout << typeid(t1).name();
\
          cout << " is before ";
\
          cout << typeid(t2).name();
\
    }
\
    else if (typeid(t2).before(typeid(t1))) {
\
          cout << typeid(t2).name();
\
          cout << " is before ";
\
          cout << typeid(t1).name();
\
    }
\
    else
\
          cout << "strange before";
\
    cout << "\n------------------------" << endl;
\
}
```

בדרך כלל, מומלץ שלא להשתמש במאקרו, אלא אם אין כל דרך אחרת לפתרון הבעיה. בהמשך נראה שבמקרה זה יש אפשרות אחרת, אך ננצל הזדמנות זו כדי להראות כיצד מגדירים ומשתמשים במאקרו מסובך. מאקרו זה דומה לפונקציה, הוא מקבל שני ארגומנטים. בניגוד לפונקציה, במאקרו מטפל הקדם-מעבד ולא המהדר. הקדם-מעבד מחליף את הארגומנטים הפורמליים של המאקרו בארגומנטים אקטואליים, שהם הארגומנטים בנקודת השימוש במאקרו. במאקרו המשתרע על יותר משורה אחת חייבים להכניס את התו \ לפני כל סוף שורה, כדי לבטל סימן זה. מאקרו משתרע תמיד על שורה **אחת**.

הפונקציה test_types מפעילה את האופרטור typeid על סוגים בסיסיים של השפה ועל מחלקה בסיסית (Base) ומחלקה שגזורה ממנה (Derive), כדי לבדוק שוויון בין אובייקטים.

```
// test the typeid operator
//
void test_types()
{
     Base b;
     Derive d;
     double dval;
     char cval;
     if (typeid(cval) == typeid(dval))
         cout << "oops something wrong" << endl;
     else
         cout << "typeid(cval) != typeid(dval)" << endl;
     if (typeid(b) == typeid(d))
         cout << "tpyeid(b) == typeid(d)" << endl;
     else
         cout << "typeid(b) != typeid(d)" << endl;
     cout << typeid(dval).name() << endl;
     //-----------------------------------------------
     // print type names using the typeinfo
     //-----------------------------------------------

     cout << "Base=" << typeid(b).name() << endl;
     cout << "Derive=" << typeid(d).name() << endl;
     cout << "double=" << typeid(double).name() << endl;
     cout << "char=" << typeid(char).name() << endl;

     display_before(b, d);
     display_before(b, char);
     display_before(b, double);
     display_before(double, Derive);
```

```
    // checking dynamic cast
    if (dynamic_cast<Base*>(&d))
        cout << "castdown has worked" << endl;
}

int main()
{
    test_types();
    return 1;
}
```

קטע התוכנית זה ידפיס את הפלט הבא:

```
typeid(cval) != typeid(dval)
typeid(b) != typeid(d)
*bp1 != *bp2 !!!
double
Base=Base
Derive=Derive
double=double
char=char
Base is before Derive
------------------------
Base is before char
------------------------
Base is before double
------------------------
Derive is before double
------------------------
castdown has worked
```

הפלט מראה לנו שמשתנים מסוגים שונים אכן מזוהים על ידי class_info. כמו כן, מודפסים כהלכה שמות המחלקות. הפלט מראה שהסדר המושג בין האובייקטים השונים הוא סדר לקסיקוגרפי בין שמות המחלקות שלה.

אפשר לכתוב פונקציה שמקבלת ייחוס לאובייקטים מסוג type_info ומשתמשת בהם כמו באובייקטים רגילים. למשל:

```
void display_info(const Type_info &t1, const Type_info &t2)
{
    if (t1.before(t2)) {
        cout << t1.name();
        cout << " is before ";
        cout << t2.name();
    }
    else if (t2.before(t1)) {
        cout << t2.name();
```

```
        cout << " is before ";
        cout << t1.name();
    }
    else
        cout << "strange before";
    cout << "\n-----------------------" << endl;

}
```

אופן השימוש בפונקציה כזו:

```
display_info(typeid(Base), typeid(Derive));
display_info(typeid(double), typeid(Derive));
display_info(typeid(char), typeid(Base));
```

האובייקטים המוחזרים על ידי typeid הם כמו אובייקטים רגילים של השפה שניתן להשתמש בהם בפונקציות של השפה. המשפט האחרון בפונקציה test_types משתמש באופרטור dynamic_cast, שהוא נושא הסעיף הבא.

## 9.1.3 האופרטור dynamic_cast

אופרטור זה מאפשר **המרה דינמית** (dynamic cast) בין מצביעים, או ייחוס, בזמן ריצה. ניתן להשתמש באופרטור מסוג מצביע למחלקה בסיסית, שמצביע בפועל לאובייקטים השייכים למחלקות הגזורות ממחלקה הבסיסית, כדי להמיר את המצביעים למצביעים המתאימים לאובייקט בפועל.

אופרטור זה יכול לשמש רק על אובייקטים פולימורפיים, על אובייקטים שיש להם לפחות פונקציה וירטואלית אחת. ההבחנה בין סוגים שונים של אובייקטים חשובה כשהאובייקטים הם פולימורפים.

אופרטור זה "מאפשר" למתכנת לפגום במערכת תוכנה פולימורפית. במילים אחרות, אופרטור זה פועל בניגוד לכללים המנחים תכנות מוכוון אובייקטים. הוא חושף את התוכנית לסוגים הקונקרטים של אובייקטים, וכך חושף את התוכנה לשינויים כשסוגי האובייקטים משתנים. בתכנות מוכוון אובייקטים מטפלת התוכנה במצביע, או ייחוס, למחלקה בסיסית מבלי לדעת את הסוג האמיתי של האובייקט. התוכנה מניחה למערכת הפולימורפית לבצע את הפונקציות המתאימות לפי סוג האובייקט האקטואלי.

בפרקים הקודמים ראינו דוגמה של ציור צורות. היתה לנו מחלקה Shape ומחלקות אחרות היורשות ממחלקה זו. אפשר לממש פונקציית ציור בדרך זו:

```
void draw(list<Shape*> &shapes)
{
   list<Shape*>::iterator i(shapes.begin());
   while (i) {
     Rectangle *rp = dynamic_cast<Rectangle*>(*i);
```

```
        if (rp) {
            // perform specific task to rectangle
        }
        else
            (*i)->draw();
    }
}
```

בפונקציה draw נבחין במקרה המיוחד של מלבן. אובייקט מסוג זה מצוירים בצורה שונה. גישה טובה יותר היא לשנות את פונקציית הציור של אובייקטים מסוג מלבן, כך שתבצע את הפעילות הנוספת הנדרשת, ולא ליצור סוגי אובייקטים חדשים בזמן ריצה.

האופרטור **dynamic_cast** פועל בצורה דומה לתבניות. בתוך ה-"<>" מכניסים את הסוג שרוצים לקבל כתוצאה מההמרה. האופרטור מקבל כארגומנט את המצביע, או את הייחוס שרוצים להמיר. ייתכנו המרות ממצביעים למצביעים של אובייקטים יורשים, או מייחוס לייחוס של אובייקט יורש.

כשממירים מצביעים, יחזיר האופרטור מצביע מבוקש לאובייקט, אם האובייקט אכן מהסוג המבוקש, או מסוג היורש מהסוג המבוקש. ההמרות נעשות בתנאי שהמרה הינה אל האובייקט של המחלקה היורשת, בצורה ציבורית, מהסוג הבסיסי. כלומר, זוהי ירושה ציבורית, המסומנת על ידי מילת המפתח public. אם ההמרה אינה מצליחה, מוחזר מצביע לאפס. קטע הקוד המשתמש באופרטור זה צריך לבדוק אם המצביע אינו מצביע ריק (NULL).

כשממירים ייחוס, אין אפשרות לבדוק אם הוא ריק. לכן, אם ההמרה נכשלת, מתעורר מצב חריג **bad_cast**. המצב מתעורר על ידי המערכת ומתבצע בזמן ריצה:

```
Base b, *bp, *bp1;
Derive d;
bp = &d;
bp1 = &b;
try {
    Derive &dr1 = dynamic_cast<Derive&>(*bp);
    Derive &dr2 = dynamic_cast<Derive&>(*bp1);
    cout << dr1.get_class_name() << "\n"
         << dr2.get_class_name();
}
catch (Bad_cast &) {
    cout << "caught bad cast in test2" << endl;
}
```

ההמרה של הייחוס ל-dr2 נכשלת, ולכן מתעורר על ידי המערכת מצב חריג **Bad_cast**. המלצתי לקורא, לא להשתמש בהמרות של ייחוס, כי הן יכולות לגרום למצב חריג. עם זאת, מומלץ גם לא להשתמש **באופרטור ההמרה**, משום ששימוש כזה **נוגד** את רוח התכנות מכוון האובייקטים. מכיון שההמרה לסוג מסוים מחייבת לדעת מהו סוג היעד, אי אפשר להשתמש בקוד כזה עבור סוגים שונים ובלתי ידועים מראש.

## 9.2 המרות נוספות של מצביעים

עד כה בוצעו המרות המצביעים על פי התחביר של C. אם היה לנו מצביע שמכיל בתוכו const יכולנו לבטל את הקביעות שלו בדרך הבאה:

```
void f(const char *cstr)
{
      char *str = (char*)cstr;
      //...
}
```

ההמרות הנוספות של מצביעים אלה פועלות בדרך הישנה ואינן מוצגות בקוד. המרה מסוג (type) אינה מצביעה על סוג ההמרה שבוצעה. לדוגמה:

```
const X *px = new X;
//...
py = (Y*)px;
```

האם ההמרה הקודמת היא המרה ממחלקה בסיסית למחלקה יורשת? האם המרה זו מבטלת את קביעות (const) המצביע? לכן, הקורא קוד כזה אינו יודע מה היתה כוונת מחבר הקוד כשביצע את המרה.

בנוסף לבעיית קריאות הקוד, אין המהדר יכול לעזור בהודעות שגיאה כשהמתכנת כותב המרה שאינה מתאימה למה שהוא מתכוון, כי אין ציון מפורש של סוג ההמרה המבוקשת.

כדי להתגבר על הבעיות שתוארו כאן, מגדירים שלושה סוגי המרות כאלה:

- **static_cast** - המרות סטטיות של מצביעים מסוג אחד לאחר.
- **const_cast** - המרות המבטלות קביעות של אובייקט.
- **reinterpert_cast** - המרות המפרשות מצביעים באופן שונה.

בסעיפים הבאים נסקור המרות אלו.

### 9.2.1 המרות סטטיות static_cast

האופרטור **static_cast<T>(e)** שקול להמרה בסגנון הישן של (T)e, המרה של ביטוי e לסוג T. בדרך כלל, **ההמרות הן ממחלקות בסיסיות למחלקות יורשות**. לדוגמה:

```
class B {...};
class D {...};
void f(B *pb, D *pd)
{
      D *pd2 = static_cast<D*>(pb);  // cast base ptr to derive
      //...
}
```

אפשר לחשוב על אופרטור זה כהמרה הפוכה של המרה הבנויה בשפה, כגון המרה ממצביע למחלקה יורשת למחלקה בסיסית. אופרטור זה שומר, קודם כל, על הקביעות, ולפיכך אין אפשרות לבטל const בעזרת אופרטור זה.

בניגוד לסגנון ההמרות הישן, ההגדרה של המחלקות המשתתפות בהמרות חייבת להיות מלאה ולא חלקית. כלומר, **ההצהרה** של מחלקות המשתתפות בהמרה חייבת להיות מלאה. בשל כך, קטע הקוד הבא אינו קביל:

```
class B;
class D1;
class D2;
B *bp;
...
D1 *dp = static_cast<D1*>(bp);
```

כשממירים ממחלקה בסיסית למחלקה יורשת, יש מקרים בהם המהדר ממיר את כתובת האובייקט. במקרים אלה חייב המהדר לדעת את גודלם של האובייקטים, לכן המרות כאלה אינן מצליחות. ההמרות בסגנון הישן רק ממירות את סוג המצביע, ללא שינוי הכתובת. מצב זה יכול לגרום לבעיה. למשל:

```
class B { ... };
class D1 : public B {...};
class D2 : public B {...};
B *bp;
...
D2 *dp = static_cast<D2*>(bp);
```

במקרה זה הערך של bp אינו מועבר ישירות ל-dp, וברוב המהדרים מתוסף אליו גודלו של אובייקט מסוג D1. לכן, המהדר מגלה בעיות כשאין הגדרה מושלמת של האובייקטים ומאפשר לקבל דיווחים מתאימים.

ההמרות של האופרטור static_cast נעשות **בזמן הידור** ולא בזמן ריצה. לכן, אם המצביע אינו מתייחס לאובייקט מתאים בזמן ריצה, עלולות להיות בעיות. במקרה כזה התנהגות התוכנית אינה מוגדרת ואינה צפויה.

## 9.2.2 המרות המבטלות קביעות const_cast

המרות שמבטלות קביעות של אובייקטים מבטלות את מילת המפתח const. המרות כאלו מבוצעות תוך שימוש במילת המפתח **const_cast**. הן יכולות להיות שימושיות באופן הבא:

```
int strcmp(char *s1, char *s2);

int const_strcmp(const char *s1, const char *s2)
{
      return (strcmp(const_cast<char*>(s1),
      const_cast<char*>(s2)));
}
```

מצביע לתווים קבועים יכול להיות מומר למצביע לתווים שאינם קבועים. שאר ההמרות שומרות על קביעות המשתנים. איני ממליץ להשתמש בהמרות המבטלות את הקביעות, מכיון שהן עלולות לגרום לבעיות בלתי צפויות. למשל:

```
const char *const const_string = "abc";

void to_upper(char *str)
{
      int d = 'A' - 'a';
      while (*str) {
            *str = *str + d;
            str++;
      }
}

to_upper(const_cast<char*>(const_string));
```

אם המהדר הקצה את המחרוזת "abc" באזור זיכרון קבוע, אזי הקריאה לפונקציה to_upper אינה מוגדרת. ביטול קביעות מראה על תחביר שגוי בתוכנית, או על הגדרה לא נכונה של פונקציות (למשל הגדרה ללא const).

### 9.2.3 המרה חוזרת reinterpret_cast

המרות המבוצעות על ידי האופרטור **reinterpret_cast** גורמות בסופו של דבר להתייחסות שונה לאזור הזיכרון. אם יש צורך לשנות ערכי מצביעים, אין להשתמש באופרטור זה. למשל, המרה ממצביע למחלקה בסיסית למצביע למחלקה יורשת אינה יכולה להתבצע בצורה טובה על ידי אופרטור זה. היא אינה משנה את ערכי המצביעים ולכן, במקרה של ירושה מרובה לא מוגדרת תוכנית המשתמשת בהמרה כזו.

ההמרה **reinterpret_cast<T>(e)** שקולה להמרה (T)e. ההבדל הוא, שכאן אין שינוי מצביעים. המרה זו טובה כשממירים ממצביעים לסוגים בסיסיים של השפה, כגון מצביעים לתווים למצביעים לשלמים. אפשר להשתמש בהמרה זו גם במצביעים לאובייקטים שאינם שייכים למחלקות הקשורות ביניהן ביחסי ירושה. למשל:

```
class X;
class Y;
void func(int *pi, chat *pc, X *py, Y *py, int ival)
{
      X *x = reinterpret_cast<X*>(pi);
      Y *y = reinterpret_cast<Y*>(px);
      char *c = reinterpret_cast<char*>(py);
      int  *ip = reinterpret_cast<int*>(pc);
      int i1 = reinterpret_cast<int>(pc);
}
```

האופרטור הזה מרשה להמיר מצביע כלשהו לכל מצביע אחר. האופרטור **reinterpret_cast** מרשה המרה של מצביעים לערכים שלמים, ולהיפך. כל ההמרות האלו הן המרות שאינן בטוחות, או שהן תלויות בסוג המהדר. כשמשתמשים בהן, האחריות לדעת את סוג המשתנים המשתתפים בהמרה היא של המתכנת.

שלא כמו האופרטור **static_cast**, אין תוצאות אופרטור זה בטוחות לשימוש. הדבר הבטוח בשפה הוא להסב את תוצאות ההמרה לסוג המקורי, ואז להשתמש בה. השפה מאפשרת להשתמש באופרטור זה כדי להמיר מצביעים לפונקציות, או מצביעים לשדות המחלקה, דבר שאינו ניתן לביצוע על ידי האופרטורים הקודמים. למשל:

```
void change_string(char *p) { *p = 'a'; }

typedef void (*PF)(const char *);

PF pf;

void f(const char *p)
{
      change_string(p);      // error "const"
      pf = change_string;    // error bad pointer type
      pf = static_cast<PF>(change_string);
                             // error bad pointer type
      pf = reinterpret_cast<PF>(change_string);
      (*pf)(p);              // oops change the string p!!!
}
```

הפעולה של השמת המצביע לפונקציה במצביע לפונקציה change_string, היא המרה מסוכנת, משום שהיא עוקפת את מערכת בדיקת ארגומנטים של פונקציות. ההמרה היחידה המותרת בשפה (פרט לסגנון הישן) היא זו המשתמשת באופרטור reinterpret_cast.

השורה האחרונה היא השורה היחידה בפונקציה האחרונה שאינה מסומנת כשגיאת הידור. גם במקרה זה ההמרה אינה בטוחה. היא גורמת לשינוי פרמטר הפונקציה f, למרות שהפונקציה מגדירה את הפרמטר כמי שאינו משתנה על ידה.

לסיכום סעיף ההמרות, ההמלצה היא להשתמש בהמרות בתבונה בכל מקרה שחייבים לעשותן. כשמשתמשים בהמרות רצוי לא להשתמש ב-reinterpret_cast. ההמרה היחידה האפשרית ללא חשש היא **static_cast**. בשאר המקרים חייבת להיות סיבה טובה מאוד לשימוש בהמרות. אני ממליץ שקורא יבדוק את עיצוב התוכנית בכל מקרה שהוא נאלץ להשתמש בהמרות.

## 9.3 מרחבי שמות namespace

ייתכנו מצבים בהם אנו מקבלים שתי ספריות, שכל אחת מהן מגדירה מחלקה מסוימת. היינו רוצים להשתמש בשתי הספריות, אך השימוש בשתיהן באותו קטע קוד גורם להתנגשויות בין השמות השונים. למשל, מספר ספריות מגדירות מחלקות כמו: Object, String.

```
// liba
class String { ... };
class Object { ... };
class Widget { ... };

// libb
class Gadget { ... };
class Object { ... };
class String { ... };
```

קטע קוד המשתמש בספריות a ו-b ייתקל בבעיית שמות. אין אפשרות להבחין בין המחלקה String מספרייה a לזו של ספרייה b.

בסעיף זה נשקול מספר אפשרויות שונות לטיפול בבעיות אלו.

### 9.3.1 שימוש בהגדרות מקוננות

פתרון פשוט, שאינו דורש תמיכה מהשפה, הוא להגדיר את המחלקות של ספרייה כלשהי בצורה מקוננת (nested). המחלקות של ספרייה X כלשהי יוגדרו בתוך רשומה בדרך זו:

```
struct libX {
      class String { ... };
      class Object { ... };
      class Widget { ... };
};
```

המתכנת המפתח את הספרייה X יכתוב כעת את הקוד שלו בדרך הבאה:

```
int libX::String::len() const
{
      return strlen(str);
}
```

המשתמש בספרייה X צריך כעת להשתמש בספרייה כך:

```
libX::String xstring("hello");
```

גישה זו אכן פועלת, אך גורמת לסרבול גם עבור מפתח הספרייה וגם עבור המתכנת המשתמש בה. כדי להתגבר על בעיה זו מספקת השפה תמיכה למקרים כאלה.

## 9.3.1.1 המונח namespace

המונח **מרחב השמות** - **namespace** - נוצר כדי להתגבר על בעיית ההתנגשויות בין מחלקות זהות בשם, לבין חלקי תוכנה שונים שצריכים לפעול באותה תוכנית. ההגדרה משייכת שמות לקבוצה, מרחב השמות, וכך אפשר להבדיל בין שמות זהים בקבוצות שונות. למשל, ניתן להגדיר כך את הפונקציות הזהות בספריות a ו-b:

```
// liba name space
namespace liba {
      class String { ... };
      class Object { ... };
      class Widget { ... };
      void f(int);
}

// libb name space
namespace libb {
      class Gadget { ... };
      class Object { ... };
      class String { ... };
      int g(char);
}
```

המתכנת המשתמש בספריות אלו יכול להשתמש בדרך הבאה בפונקציות ואובייקטים מספריות אלו:

```
liba::String s1("hello");
libb::String s2("world");

void g(int i, char c)
{
      liba::f(i);
      libb::g(c);
}
```

המתכנת משתמש בספריות אלו בצורה הבסיסית, כמו במחלקות מקוננות. כשרוצים להשתמש בפונקציה, או במחלקה, השייכת למרחב שמות כלשהו, עלינו להשתמש באופרטור "::" שקובע את מרחב השמות האקטואלי שמתייחסים אליו.

השורות הראשונות מגדירות אובייקטים מסוג מחרוזות; הראשון ביניהם שייך לספרייה liba והשני שייך לספרייה libb. בצורה דומה משתמשת הפונקציה g בפונקציות f ו-g מהספריות liba ו-libb, בהתאמה.

ההמתכנת יכתוב את הפונקציות והאובייקטים של הספריות באופן הבא:

```
liba::String::String(const char *s)
{
      //...
}
```

```
int libb::g(char c)
{
      //...
}
```

גם כאן משתמשים באופרטור "::" כדי לשייך פונקציות, או אובייקטים למרחב שמות. דוגמה זו מגדירה את בנאי המחלקה String במרחב השמות liba ואת הפונקציה g - במרחב השמות libg.

## 9.3.1.2 שימוש במילת המפתח using

עד כה ראינו שימוש במרחב שמות שהיה שקול לשימוש במחלקות מקוננות. גם ראינו שיש להשתמש באופרטור "::" כדי לשייך אובייקטים, או פונקציות למרחב השמות. מצב זה אינו נוח ומסרבל את הקוד, הן של המשתמש והן של כותב הספרייה.

פתרון הבעיה הוא מילת המפתח **using**. השימוש בה גורם להכרת שמות של מרחב השמות מאותה נקודה ואילך, ללא צורך בציון מפורש של מרחב השמות. יש שתי צורות לשימוש במילת המפתח using. בדרך השימוש הראשונה גורמים להכרת כל השמות במרחב השמות, ובשנייה גורמים להכרה בצורה בדידה של פונקציות, או אובייקטים.

### 9.3.1.2.1 שימוש מוכלל

כשמשתמשים בשימוש המוכלל, גורמים לכך שכל השמות של מרחב הכתובות מוכרים מעתה בקטע הקוד המשתמש בהם. דבר זה מתבצע כך:

```
using namespace liba;

String s1("hello");      // using liba::String
int i;
f(i);                    // using liba::f
```

כשמשתמשים במילות המפתח using namespace "מכניסים" את האובייקטים והפונקציות של מרחב השמות לתחום המתאים.

השימוש באובייקטים או בפונקציות של הספרייה liba, אינו דורש שימוש בשם מרחב הכתובות. מצב זה מקל על השימוש במרחב השמות, אך גורם לבעיות כשיש שמות זהים לאובייקטים או פונקציות ממרחבי שמות שונים.

```
using namespace liba;
using namespace libb;

String s("world");
```

במצב כזה אין אפשרות להבחין באיזה אובייקט להשתמש מבין מרחבי השמות השונים. במקרה כזה יש להשתמש באופן מוכלל **רק באחד** ממרחבי השמות שעליו

מצהירים עם using. אובייקטים או פונקציות של מרחבי שמות אחרים יש לציין **באופן מפורש**. למשל:

```
using namespace liba;

String s("hello");
libb::String s1("world");
```

#### 9.3.1.2.2 שימוש מפורש

בציון מפורש של מרחב שמות נעשה באמצעות מילת המפתח using בצורה בדידה לאובייקטים, או לפונקציות ממרחב שמות נתון. תחביר השימוש המפורש:

```
using liba::String;
using libb::g;
```

בשורות אלו מציינים שהמחלקה String שייכת לספרייה liba והפונקציה g שייכת לספרייה libb. לאחר הגדרות אלו אין צורך להשתמש בשמות מרחבי השמות. אפשר להשתמש באובייקט ובפונקציה באופן ישיר, בצורה הבאה:

```
String s("a");     // using liba
char c;
g(c);              // using liba
```

השימוש במילת המפתח using גרם להכרת הספריות של השמות והאובייקטים שאנו מתייחסים אליהם. אם מוגדרים כבר בתחום זה משתנים או מחלקות באותם שמות, נוצרת התנגשות שמות והמהדר יסמן זאת כשגיאה.

## 9.3.2 כינון של מרחבי שמות

ניתן לקונן מרחבי שמות כפי שאפשר לקונן מחלקות. קינון מרחבי שמות מיועד למרחבים מורכבים, שבהם ממספר מרחבי שמות. הגדרת מרחבי שמות מקוננים נעשית בדרך זו:

```
namespace Outer {
      void f1();
      // ...
      namespace Inner {
            void f2();
            void f3();
            // ...
      }
}
```

השימוש ב**מרחבי שמות מקוננים** (nested namespace) דומה לשימוש במחלקות מקוננות. כלומר, יש לספק את השמות של כל מרחבי השמות, מהחיצוני לפנימי, באופן הבא:

```
void Outer::f1()
{
...
}

void Outer::Inner::f2()
{
	f1();
}
```

בפונקציות שנמצאות במרחב שמות פנימי אין צורך להשתמש בשם של מרחב השמות החיצוני, או הפנימי. לא כך הדבר לגבי פונקציות השייכות למרחב השמות החיצוני. פונקציות כאלו יכולות להשתמש בפונקציות מאותו מרחב בצורה רגילה, אך עליהן להשתמש בשם מרחב השמות בצורה מפורשת בעת קריאה לפונקציות של מרחב השמות הפנימי. למשל, הפונקציה f1 חייבת לציין את שם מרחב השמות המקונן כשהיא משתמשת בפונקציה של מרחב השמות הפנימי (Inner).

```
void Outer::f1()
{
	Inner::f3();
}
```

## 9.3.3 הרחבת מרחב שמות

אפשר להגדיל **מרחב שמות** (namespace) על ידי הוספת פונקציות, או אובייקטים. אפשרות זו אינה נתונה למחלקות. במרחב שמות "**פתוח**" אפשר תמיד להוסיף שמות. הדבר נעשה כך:

```
namespace A {
	void f();
	class Widget { ... };
};

// ...

A::f();
A::g();		// error undefined function
namespace A {
	void g();	// add g to name space A
}

A::g();		// OK g has been added to the namespace
```

הוספת פונקציות או אובייקטים יכולה להיות בכל מקום ובכל סדר. לכן, הקריאה הראשונה לפונקציה g אינה מוגדרת. לעומת זאת, הקריאה השנייה לפונקציה g מוגדרת, שכן הפונקציה הוספה למרחב השמות.

## 9.4 מילת המפתח mutable

לשפת התכנות התוספה מילת מפתח **mutable**, אשר מתייחסת לשדות שונים באובייקטים. כשמציינים ששדה מסוים באובייקט הוא מסוג **mutable**, המשמעות היא שפונקציות קבועות של האובייקט יכולות לשנות אותו. כלומר:

```
class X {
      mutable int j;
      int k;
public:
      X() { k = 0; j = 0; }
      void set_j(int jo) const
      { j = jo; }
      void set_k(void ko)
      { k = ko; }
};
```

הפונקציה (set_j), המשנה את ערך השדה j, יכולה להיות קבועה, כלומר, אינה משנה את האובייקט עליו היא פועלת. לעומת זאת, הפונקציה (set_k) שמשנה שדה רגיל (שאינו mutable) אינה יכולה להיות קבועה. אפשר להשתמש בפונקציות אלו בדרך זו:

```
const X xc;
X x;
x.set_j(1);
x.set_k(2);
xc.set_j(3);      // ok - j is mutable
xc.set_k(4);      // compilation error - xc in constant
```

רק הפונקציה הקבועה (set_j) מותרת להפעלה על האובייקט הקבוע xc. הפונקציה השנייה (set_k) אינה קבועה, ולכן המהדר דוחה את ההפעלה שלה על האובייקט הקבוע xc.

תכונות **שדה שניתן לשינוי** (mutable) באובייקטים קבועים, נראים תמוהים במבט ראשון, אך יש בהם היגיון. כדי להבין זאת, נתבונן בדוגמה. נניח, שיש לנו רשימה שמאפשרת להכניס לתוכה, או להוציא ממנה אובייקטים. כמו כן, היא מאפשרת לסרוק אותה. לצורך האיטרציה (כלומר, המעבר על פני הרשימה) מחזיקה הרשימה מצביע לצומת הנוכחי. שינוי מקום האיטרציה אינו משנה את הרשימה מבחינה לוגית. כלומר, הרשימה אינה קטנה או גדלה. דרישה הגיונית נוספת היא לאפשר למשתמש ברשימה לעבור על רשימה קבועה. כדי להשיג זאת, יכול מפתח הרשימה להשתמש במילת המפתח **mutable** באופן הבא:

```
class List {
   struct node {
      void *data;
      node *next;
   };
   node *head;           // head of the list
```

```
  mutable node *current;// current iteration node
public:
  List() { current = head = 0; }
  void *first()
  { current = head; return (current ? current->data : 0); }
  const void *first() const
  { current = head; return (current ? current->data : 0); }
  void *next()
  { if (current) current = current->next;
    return (current ? current->data : 0); }
  const void *next() const
  { if (current) current = current->next;
    return (current ? current->data : 0); }
  ...
};
```

המתכנת שמפתח את המחלקה מגדיר שתי גרסאות של פונקציות שמבצעות איטרציות על הרשימה. גירסה ראשונה היא של פונקציות שאינן קבועות, והגירסה השנייה היא של פונקציות קבועות. במילים אחרות, הגירסה השנייה אינה משנה את האובייקט עליו היא פועלת. לפיכך, אפשר לבצע איטרציה על רשימה קבועה.

## 9.5 מילת מפתח explicit (מפורש)

**בנאים** של מחלקות בעלי ארגומנט יחיד יכולים לשמש כ**אופרטורי המרה**. כלומר, בנאים יכולים לשמש כאופרטורי המרה מסוג הארגומנט שלהם אל אובייקט המחלקה. כשיש פונקציות המקבלות אובייקט, רשאי המהדר להמיר את הארגומנט האקטואלי לאובייקט של המחלקה. למשל:

```
class Text {
      char *str;
public:
      Text(const char *s);
      ...
};
void f(const Text &t)
{
      ...
}
f("This is a string");
```

הפעלת הפונקציה f גורמת ליצירת אובייקט זמני על ידי המהדר. אובייקט זה הוא מסוג Text. האובייקט נמסר לפונקציה f והמהדר דואג להרוס אותו אחרי שהפונקציה חוזרת. לפעמים מצב זה אינו מה שרוצה מפתח המחלקה Text, כי יצירת אובייקט זמני עשויה להשפיע על זמני הריצה של התוכנית. כדי **למנוע** המרה אוטומטית על ידי המהדר, אפשר להשתמש במילת המפתח **explicit**. שימוש במילה זו מסמן למהדר שלא

להשתמש בבנאי ההמרה, אלא אם נורה לו אחרת בצורה מפורשת. לכן, אפשר להשתמש במילת המפתח בצורה הבאה:

```
class X {
      char *str;
public:
      Text(const char *s) explicit;
      ...
};

void f(const Text &t)
{
      ...
}

f("This is a string");  //Error only explict use of
                        // constructor is allowed
f(Text("This is a string"));
                        // OK conversion is used explicity
```

בקריאה השנייה לפונקציה f מתאפשרת ההמרה באופן מפורש, ולעומתה נכשלת הקריאה הראשונה כבר בשלב הידור התוכנית. צורה זו ברורה יותר למתכנת שמשתמש במחלקה Text. בנוסף, המתכנת יכול להחליט מתי לבצע את ההמרה בצורה מפורשת, ואז לשלם ביצירת אובייקטים זמניים.

## 9.6 סיכום

בפרק זה ראינו אפשרויות רבות וחדשות שהתוספו לשפת התכנות ++C בעת האחרונה, ולכן אינן נתמכות עדיין על ידי מהדרים רבים. **הדוגמאות בפרק זה יכולות לפעול עם חלק מהמהדרים בלבד, לא עם כולם!** עם זאת לא ירחק היום שתכונות אלו יהיו חלק אינטגרלי של כל מהדר ++C.

**מידע בזמן ריצה** (RTTI) היא היכולת לקבל את סוג האובייקט בזמן ריצה. במערכת פולימורפית בה מטפלים באובייקטים בעזרת מצביעים גנרים למחלקות בסיסיות יש לכך חשיבות רבה. מידע בזמן ריצה יכול לעזור כשיש לבצע פעולות שונות כדוגמת השוואה בין אובייקטים שונים.

סגנון חדש של המרות מצביעים, כגון **dynamic_cast**, **static_cast**, **const_cast** ו-**reintrepert_cast**, תורם ליכולת המהדר לגלות שגיאות, עוד בזמן קומפילציה ולדווח עליהן למתכנת. בנוסף, שימוש באופרטורים אלה גורם לתוכנה להיות הרבה יותר ברורה, כשהקורא את התוכנה מבין בדיוק מה מחבר הפונקציה רצה לבצע.

**מרחבי שמות** (namespace) באים לפתור התנגשויות בין שמות בחלקים בלתי תלויים של התוכנה. השימוש במרחב שמות מזכיר את מרחב המחלקה. לעומת מרחב

המחלקה, שבו יש להשתמש באופרטור ההבחנה "::", לכל פעולה אפשר להביא שמות ממרחב שמות אחד לשני, כך שאין צורך להשתמש באופרטור ההבחנה. מרחבי שמות הם פתוחים, כלומר, ניתן להרחיבם במספר מקומות. מחלקות לעומת זאת, אינן פתוחות ולא ניתנות להרחבה.

## 9.7 שאלות

1. כתוב תוכנית המאפשרת לכתוב ולקרוא אובייקטים לדיסק ומהדיסק.
2. השתמש במידע בזמן ריצה כדי לכתוב פונקציה כללית המשווה בין אובייקטים פולימורפיים.
3. ממש את הרשימה ואת האיטרטורים מהפרק הקודם, בעזרת מרחבי שמות.

# פרק 10

# ספריית תבניות סטנדרטיות STL

בפרק זה, ובפרקים הבאים, נלמד על **ספריית התבניות הסטנדרטית** - **STL** (Standard Template Library) של ++C. פרק זה משמש כמבוא ותיאור כללי של ספרייה זו, וגם נסביר בו את עקרונות התכנון של הספרייה. בפרקים הבאים נראה מספר דוגמאות ונסקור מחלקות שונות של הספרייה.

בספרייה יש שימוש נרחב בתבניות ++C. למרות שספרייה זו היא סטנדרטית והתקבלה על ידי המועצה של **ANSI**, מרבית המהדרים עדיין אינם תומכים בה. בעת כתיבת ספר זה תומכים בספרייה המהדרים של בורלנד מגרסאות 4.x ומעלה, המהדר של GNU (++g), והמהדר ++Visual C של מיקרוסופט, מגירסה 4 ומעלה. חברת Sun הכריזה על מהדר שיתמוך גם הוא בספרייה זו. הדוגמאות בפרק זה הורצו בעזרת המהדרים של **בורלנד**.

## 10.1 עקרונות הספרייה STL

הספרייה הסטנדרטית STL מורכבת ברובה ממחלקות תבנית ופונקציות תבנית. בספרייה יש שימוש נרחב בתבניות ולכן נדרש ידע בסיסי לפחות בנושא, אם לא למעלה מזה. קורא ששקד וקרא את הפרקים הקודמים, רכש ידע מספק כדי ל"התמודד" עם השימוש בספרייה זו.

הספרייה אינה מתבססת על ירושה, אלא על **תכנות גנרי** (generic programming). ראינו לכך דוגמאות בפרק על תבניות ופיתחנו וקטור ורשימה מקושרת המבוססים על תבניות. פיתחנו גם אלגוריתמים המטפלים במבני נתונים שונים, ללא צורך לדעת את סוג מבנה הנתונים. תכנות גנרי משתמש בפונקציות המקבלות איטרטורים לתחום מסוים שעליהן הן עובדות. האיטרטורים מתנהגים כמו מצביעים, ולכן פונקציות גנריות טובות גם למצביעים וגם למחלקות. על עקרונות אלו מבוססת הספרייה הסטנדרטית של ++C.

הספרייה עצמה מחולקת לשלושה מרכיבים עיקריים, כשיש מספר מרכיבים משניים. מרכיבי הספרייה העיקריים הם:

- **מכולות** (Containers, או **אוספים**) - אלה הם אובייקטים שמכילים אובייקטים אחרים, כמו למשל רשימה או מערך.
- **איטרטורים** - אובייקטים מסוג זה מאפשרים איטרציות על אוספים המכילים אובייקטים אחרים.
- **אלגוריתמים** - אלה הן פונקציות תבנית שאינן קשורות במחלקה מסוימת באופן מיוחד. פונקציות אלו מבצעות אלגוריתם, כמו find, המוצא אובייקט באוסף כלשהו.

STL, אם כן, היא ספרייה של אוספי אובייקטים שהם למעשה מחלקות שנבנו בצורה גנרית, ולכן אפשר להשתמש בהן כמעט בכל יישום. בהיסטוריה של C++ כבר היו מספר לא קטן של ספריות כאלו, כמו למשל ספריית האובייקטים NIH, ספריית האובייקטים של בורלנד וספריית האובייקטים של Rogue Waves.

החיסרון של כל הספריות שהזכרנו בכך שלא היו סטנדרטיות. הספריות פעלו בסביבה מסוימת, אך כשהעבירו את התוכנה שהשתמשה בהן לסביבה אחרת, היה צורך להעביר גם את הספרייה. במקרים רבים היה הדבר קשה, ולעיתים בלתי אפשרי. ספריית STL תצורף לכל מהדר עם ממשק ANSI, ולכן, תוכנה שתיכתב בעזרת STL בסביבה מסוימת תפעל גם בסביבה אחרת.

חיסרון אחר של הספריות הקודמות הוא **אלגוריתמים**. הספריות הקודמות לא תמכו באלגוריתמים כגון sort, או unique (נראה בהמשך). הן סיפקו מבני נתונים שאיפשרו להכניס, להוציא ולמצוא אובייקטים, וגם סיפקו איטרטורים.

הגישה בספריית האובייקטים הסטנדרטית של C++ **שונה** מהגישה המקובלת בתכנות מוכוון אובייקטים. לפי גישה זו, יש מחלקה מסוימת ופעולות ששייכות למחלקה, אך אין פונקציות חיצוניות שפועלות עליה. גישה זו אכן שומרת על עיקרון הסתרת המידע של המחלקה, אך אינה מאפשרת לנצל תכונות משותפות בין משתנים בסיסיים של השפה לבין מחלקות המייצגות אוספים של אובייקטים.

גישה זו מתאימה **לשפת תכנות מוכוונות אובייקטים טהורה** (pure object oriented language), כמו Smalltalk. ב-Smalltalk כל משתנה, או מצביע לאובייקט, שנגזר מאובייקט בסיסי נקרא Object. לכן, כל פונקציה שייכת למחלקה Object או מחלקה אחרת היורשת ממנה. ממילא, כל פונקציה שפועלת על אובייקטים מסוג Object פועלת על אובייקט היורש ממנה. לכן, אין מקום וצורך בפונקציות גנריות. לדבר זה יש יתרונות וחסרונות, והדעות בנושא זה חלוקות. ספר זה עוסק ב-C++ ולא בפילוסופיה, ולכן לא נעסוק בוויכוח זה.

שפת C++ מספקת משתנים בסיסיים ופונקציות, בנוסף לתמיכה בתכנות מוכוון אובייקטים, ולכן היא נקראת C++ **שפה משולבת** (Hybrid). כשאנו כותבים פונקציית מיון ב-C++ עבור מערך של אובייקטים כמו String, נרצה להשתמש בפונקציה זו גם עבור מערך של מצביעים לתווים. איננו רוצים להעתיק את מערך המצביעים לתווים למערך של אובייקטים ולבצע את המיון רק לאחר מכן. לכן, אם נכתוב פונקציית מיון עבור מחלקה המייצגת מערך של אובייקטים, היא לא תהיה טובה עבור מערך של משתנים בסיסיים של השפה.

תכנות גנרי הוא המונח העומד מאחורי הספרייה הסטנדרטית, והוא הרעיון המרכזי בספרייה זו. בתכנות גנרי אנו מנסים להתעלם מסוג האובייקטים, או המשתנים עליהם פועל האלגוריתם, ולהתרכז בפעולה הבסיסית של האלגוריתם עצמו. למשל, אלגוריתם מיון צריך רק לקבל יחס סדר בין האובייקטים ותחום המתאר את קבוצת האובייקטים שצריך למיין.

סיבות אלו מראות מדוע ספריות האובייקטים לא הצליחו ב-C++ כמו בשפות תכנות אחרות. **STL** פותרת בעיות אלו ומספקת מבני נתונים רבים ומורכבים ביעילות רבה, וגם אלגוריתמים הפועלים על מבני נתונים אלה ועל מערכים בסיסיים של השפה.

הקוד שפותח ב-STL יעיל מאוד, לכן אין כל צורך לשכתב את הספרייה הבסיסית כדי להגיע לביצועים גבוהים יותר, פרט לבעיה אחת. הבעיה בקוד זה היא שכולו מבוסס על תבניות, ולכן כמות הקוד שיוצר המהדר היא רבה כפי שנראה בהמשך. בספר זה נשתמש בגירסה הציבורית שפותחה **בחברת HP** על ידי **Alex Stavanof**. נציין את יתרונותיה של ספרייה זו בהמשך.

ספריית אובייקטים המספקת מיגוון רב של אובייקטים מקלה באופן משמעותי על כתיבת התוכנה, חוסכת זמן וטעויות וגם יעילה מאוד. ספרייה כזו הופכת את התכנות לפעולה נעימה, וחוסכת את הצורך לשוב ולדאוג למחלקות שכבר כתבנו.

## 10.2 מכולות ב-STL

ב-STL יש מספר **מכולות** (containers) השונות זה מזה ביכולות שלהם, אשר מתאימים למצבים שונים. הממשק לאובייקטים השייכים למחלקות אלו דומה בדרך כלל לממשקים שאנו מכירים, ולכן הדבר מאפשר פיתוח אלגוריתמים שעובדים עבור מספר רב של מכולות ואינם קשורים למחלקה יחידה. במילים אחרות, האלגוריתמים אינם פונקציות של מחלקה. המחלקות המייצגות מכולות, כוללות פונקציות הכנסה והוצאה, התלויות במבנה הנתונים של המחלקה.

כל המכולות מכילות אובייקטים, ולא מצביעים לאובייקטים. כלומר, כשמכניסים אובייקט למכולה, המכולה יוצרת תחילה **העתק** של האובייקט, ואת ההעתק הזה היא מצרפת לאוסף האובייקטים שלה. מסיבה זו אין שיתוף בין אובייקטים במספר מכולות. המתכנת יכול ליצור אוסף של מצביעים ואז ליצור שיתוף בין אובייקטים במכולות שונות. במצב כזה אחראי המתכנת להקצאה ושחרור של אובייקטים. נניח, למשל, שקיימת מכולה כלשהי:

```
container<String*> c1, c2;
String *sp = new String ("ptr");
c1.push_back(sp);
c1.push_back(sp);
```

הכנסנו מצביעים למכולות ולכן המתכנת גם חייב לבטל את האובייקטים האלה. כשמוכנסים אובייקטים (ולא מצביעים), אחראי האוסף לשחרור והקצאת זיכרון עבור הצמתים שבאוסף, וגם עבור האובייקטים שצורפו לאוסף.

המכולות מאפשרות למשתמש בהן לשנות את הדרך שבה מוקצה עבורן זיכרון. כך יכול המשתמש לקבוע את אופן הקצאת הזיכרון. לפי מודל התוכנית אפשר לקבוע את סוג המצביעים ל-near או far במונחים של מחשב אישי. הדבר מאפשר גם לקבוע זיכרון שיהיה זמין גם מעבר לתקופת החיים של התוכנית, כלומר **Persistent**.

האוספים הנתמכים ב-STL הם:

| | |
|---|---|
| vector | אוסף הדומה לווקטור (דומה לזה שראינו בפרקים הקודמים), האובייקטים באוסף כזה אגורים בצורה סדרתית בזיכרון רציף. הגישה לאובייקטים באוסף כזה היא אקראית בעזרת אופרטור []. |
| list | רשימה מעגלית כפולה. הגישה לאובייקטים באוסף זה היא סדרתית, קדימה או אחורה. |
| deque | תור כפול שמאפשר להכניס אובייקטים משני הצדדים (תחילת התור וסופו). התור הכפול גם מאפשר להוציא אובייקטים משני צדדיו. |
| set | אוסף סדור (קבוצה) של אובייקטים. לאובייקטים המוכנסים לאוסף זה יש יחס של סדר. הדבר אינו דומה להגדרה המתמטית של אוסף (ההגדרה המתמטית של קבוצה היא: אוסף שאינו סדור של אובייקטים, ללא חזרות). באוסף זה אסורות החזרות של אובייקטים. התכונה החזקה של אוסף זה היא מהירות גבוהה של חיפוש אובייקטים. |
| multiset | אוסף הדומה לאוסף הקודם. אוסף סדור של אובייקטים, אך החזרות מותרות. כלומר, מותר להכניס את אותו אובייקט מספר פעמים. |
| map | מפה, לעיתים נקרא מבנה נתונים זה בעגה המקצועית - **מילון**. זהו אוסף של זוגות אובייקטים, שכל אחד מהם מורכב ממפתח וערך. המפה היא אוסף סדור לפי המפתחות. באוסף מסוג זה ייתכן רק מפתח יחיד מערך מסוים. |
| multimap | אוסף סדור של זוגות אובייקטים, כשכל זוג מורכב ממפתח ומערך. במקרה זה מותר מספר זוגות עם מפתח זהה. |

כל המכולות, או האוספים, מאפשרים להכניס כל אובייקט לתוכם, כולל סוגים בסיסיים של השפה, כמו שלמים או float.

## 10.3 איטרטורים

**איטרטורים** (iterators) הם אובייקטים שמאפשרים סריקה של אוספים, כלומר - של מכולות. לכל אחד מאוספים אלה יש הגדרה של איטרטור. איטרטור מסוג מסוים מתאים אך ורק לאוסף מסוים. דבר זה מאפשר להפעיל פונקציות בצורה גנרית, ללא כל צורך בידיעת סוג האוסף.

STL רואה את האיטרטורים כסוג מצביע מסוים. כשרוצים לקדם מצביע, מפעילים עליו אופרטור "++". כשרוצים להחזיר את המצביע לאחור מפעילים עליו את

האופרטור "--". האופרטור "מצביע" לאובייקט נוכחי שנמצא באוסף. לכל איטרטור יש מידע המציין את האובייקט הנוכחי של האיטרציה.

כשרוצים להשתמש בתכולת מצביע משתמשים באופרטור "*". בדומה, כשרוצים לקבל את האובייקט הנוכחי של האיטרציה משתמשים באופרטור "*". הדבר מאפשר לפתח אלגוריתמים שונים המתאימים למצביעים וגם לאיטרטורים. כבר נפגשנו בטכניקה זו בפרקים הקודמים של ספר זה. **STL** מגדירה את סוגי האיטרטורים הבאים:

- **Random access iterator** - **איטרטור לגישה אקראית** - איטרטור מסוג זה תומך באופרטורים "++", "--" ו-"[]" והוא בעל כל התכונות של מצביעים בסיסיים בשפה. האופרטור "++" מאפשר להתקדם באוסף שאליו מצביע האיטרטור. האופרטור "--" מאפשר לנוע לאחור באוסף שאליו מצביע האיטרטור. האופרטור "[]" מאפשר גישה לכל מקום באוסף שאליו מצביע האיטרטור. הגישה במקרה זה היא על פי אינדקס.
- **Bidirectional direction iterator** - **איטרטור דו-כיווני** - איטרטור מסוג זה מאפשר לנוע לשני הכיוונים באוסף שאליו הוא מצביע. התנועה עם איטרטור זה מתאפשרת על ידי האופרטורים "++" ו- "--". איטרטור זה אינו מאפשר גישה אקראית בעזרת אופרטור "[]". איטרטור זה נחשב לבעל יכולות נמוכות יותר מהקודם.
- **Forward iterator** - **איטרטור מתקדם** - איטרטור מסוג זה מאפשר לנוע קדימה בלבד באוסף שאליו הוא מצביע. הוא עושה זאת בעזרת האופרטור "++".
- **Backward iterator** - **איטרטור נסוג** - איטרטור מסוג זה מאפשר תנועה לאחור בלבד באוסף אליו הוא מצביע. הוא עושה זאת בעזרת האופרטור "--".
- **Input iterator** - **איטרטור קלט** - איטרטור זה מאפשר השמה בלבד של אובייקטים ממנו אל אובייקט כלשהו. בדרך כלל, הוא קשור לקובץ ומאפשר לחלץ ממנו נתונים.
- **Output iterators** - **איטרטור פלט** - איטרטור זה מאפשר השמה של אובייקטים לתוכו. בדרך כלל, האיטרטור קשור לקובץ ומשמש לפלט.

לכל אוסף **container** מוגדר האיטרטור בצורה הבאה:

```
container::iterator i;
```

דבר זה מאפשר לאלגוריתמים שונים להשתמש באיטרטור של מכולה נתונה, מבלי לדעת את סוג המכולה או את סוג האיטרטור.

## 10.3.1 שימוש באלגוריתמים ואיטרטורים

האלגוריתמים השונים יכולים לפעול גם על מצביעים בסיסיים של השפה. במקרים כאלה יש להבדיל בין סוגי האיטרטורים כדי לממש את האלגוריתמים באופן יעיל. כדי להבין את משמעות הדבר נבחן דוגמה. נניח, שיש לנו פונקציה המקדמת איטרטור, או מצביע, במספר נתון n.

```
template <class Iterator>
void advance(Iterator &iter, int n)
{
      iter += n;
}
```

הפונקציה הגנרית הזו יכולה לעבוד עבור **איטרטורים** מסוג **מצביעים**, או עם **איטרטורים** מסוג **גישה אקראית**. עבור איטרטורים דו-כיוונים אין פעולה "=+", ולכן קטע קוד זה אינו עובר הידור. אפשר לשנות את הפונקציה הזו בצורה הבאה:

```
template <Iterator>
void advance(Iterator &iter, int n)
{
    if (n > 0)
      while (n-- > 0)
            ++iter;
    else
      while (n++ > 0)
            --iter;
}
```

פונקציה זו מתאימה לפעולה עם **איטרטורים מסוג גישה אקראית** (random access iterator), או **איטרטורים דו-כיוונים**. הבעיה נוצרת כשמפעילים פונקציה זו על איטרטור דו-כיווני. במקרה כזה מספיק המשפט הזה:

```
iter += n;
```

במקרים כאלה יעילות הפונקציה גרועה במיוחד, כי נדרשות n פעולות במקום פעולה אחת. כדי לפתור בעיה זו מוגדר המושג **תווית (tag)** עבור איטרטורים. נסקור מושג זה בסעיף הבא.

## 10.3.2 תווית של איטרטור

כפי שניתן לראות, קיימת בעיית הבחנה בין הפונקציות השונות. אם נכתוב פונקציה גנרית שתקדם איטרטור, היא תהיה בזבזנית מאוד עבור איטרטורים לגישה אקראית. פתרון אחד הוא להגדיר היררכיה של מחלקות איטרטורים. לדוגמה:

```
template <class Type>
class forward_iterator {
public:
    forward_iterator() {}
    forward_iterator &operator++()
    { return *this; }
    // ...
};
template <class Type>
class random_access_iterator {
```

```
public:
    random_access_iterator() {}
    random_access_iterator &operator+=(int n);
    ...
};
```

כעת יכולנו לגזור את האיטרטורים שלנו מהאיטרטורים הבסיסיים, למשל:

```
template <class Type>
class deque {
...
public:
    class iterator : public random_access_iterator<Type> {
    ...
    iterator &operator+=(int n)
    { cur += n; return *this; }
    ...
};
```

עכשיו אפשר להשתמש ביכולת של C++ להגדיר גירסה מסוימת של הפונקציה הגנרית advance, באופן הבא:

```
template <class Type>
void advance(random_access_iterator<Type> &j, int n)
{
    j += n;
}
```

הבעיה המתגלה בפתרון זה היא בכך שהוא אינו יכול לפעול עבור מצביעים בסיסיים של השפה. הסיבה לכך היא שהמצביעים הבסיסיים אינם יורשים ממחלקה כלשהי, ובפרט אינם יורשים מהמחלקה random_access_iterator.

הפתרון שמציגה הספרייה **STL** לבעיה, הוא שימוש ב**תוויות איטרטורים** (iterator tags). תחילה מגדירים את התוויות השונות של האיטרטורים:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_acess_iterator_tag {};
```

לאחר מכן מוגדרים האיטרטורים השונים והקטגוריות הבסיסיות שלהם. קטגוריות אלו מספקות מחלקות בסיסיות עבור איטרטורים המוגדרים במכולות השונות.

```
template <class T> struct input_iterator {};
struct output_iterator {};
template <class T> struct forward_iterator {};
template <class T> struct bidirectional_iterator {};
template <class T> struct random_access_iterator {};
```

לכל איטרטור מוגדרת ה**קטגוריה** אליה הוא שייך. אין זו מחלקה בסיסית שממנה יורשים האיטרטורים, אלא הקטגוריה בלבד. כזכור, אין יחס של ירושה למצביעים הבסיסיים של השפה.

```
template <class T>
inline input_iterator_tag
iterator_category(const input_iterator<T>&) {
    return input_iterator_tag();
}

inline output_iterator_tag iterator_category(const
output_iterator&) {
    return output_iterator_tag();
}

template <class T>
inline forward_iterator_tag
iterator_category(const forward_iterator<T>&) {
    return forward_iterator_tag();
}

template <class T>
inline bidirectional_iterator_tag
iterator_category(const bidirectional_iterator<T>&) {
    return bidirectional_iterator_tag();
}

template <class T>
inline random_access_iterator_tag
iterator_category(const random_access_iterator<T>&) {
    return random_access_iterator_tag();
}

template <class T>
inline random_access_iterator_tag iterator_category(const
T*) {
    return random_access_iterator_tag();
}
```

**הערה!**

*בפונקציה ובמחלקות המקוריות יש פרמטר תבנית נוסף, Distance. לצורך פשטות ההסבר הרשיתי לעצמי לבטל אותו. הורדת הפרמטר אינה משנה את עקרון התוויות לאיטרטורים.*

**לאיטרטורים מסוג גישה אקראית** (random access iterator) יש שתי פונקציות מועמסות המספקות תוויות. הפונקציה הראשונה מספקת תווית איטרטור מסוג גישה אקראית עבור איטרטורים מסוג מחלקות, והפונקציה השנייה מספקת תווית מסוג של איטרטור גישה אקראית עבור איטרטורים שהם מצביעים בסיסיים. כעת, באמצעות תוויות האיטרטורים אנו יכולים לכתוב את הפונקציה advance בצורה הבאה:

```
template <clas Iter>
inline void advance(Iter &j, int n)
{
    _advance(j, n,iterator_category(j));
}

template <class RandomAcessIterator>
inline void _advance(RandomAcessIterator &j,
                     int n, random_access_iterator_tag)
{
    j += n;
}

template <class BidirctionalItrator>
void _advance(BidirectionalIterator &j,
        int n, bidirectional_iterator_tag)
{
    if (n >= 0)
      while (n--)
           ++j;
    else
      while (n++)
           --j;
}
```

כלומר, הפונקציה הראשית שנקראת מפנה את הקריאה לפונקציה אחרת, שלה יש את אותם הארגומנטים, בתוספת תווית זיהוי של האיטרטור. מנגנון העמסה של C++ בוחר את הפונקציה המתאימה לפי סוג התווית! טכניקה זו מאפשרת הפעלה של פונקציות שונות בצורה **פולימורפית סטטית** (static polymorphism). סוג הפונקציה נבחר על ידי המהדר לפי סוג הפרמטרים (התווית של האיטרטור) **בזמן הידור**, ולא בזמן ריצה.

# 10.4 אלגוריתמים

אחד הדברים המייחדים את STL הם ה**אלגוריתמים** של ספרייה זו. בניגוד לספריות אחרות, מספקת STL מיגוון רחב של אלגוריתמים גנרים המתאימים ליישומים ושימושים רבים. אלגוריתמים רבים מתוכם טובים גם לאוספים בסיסיים של השפה, כמו **מערכים**.

אפשר לקחת את האלגוריתמים של STL ולהשתמש בהם עבור ספריות אחרות, שחסרות אלגוריתמים כאלה. למשל, ספריית האובייקטים של Rogue Waves החדשה מאפשרת שימוש באלגוריתמים של STL.

בדרך כלל, האלגוריתמים ב-STL פועלים על תחום נתון. העיקרון המנחה הוא פעולה בעזרת איטרטורים. האלגוריתם מקבל, בדרך כלל, איטרטור להתחלת התחום ואיטרטור לסוף התחום. אלגוריתם כזה, למשל, הוא find המוצא אובייקט נתון בתחום. האלגוריתם ממומש בצורה הבאה:

```
template <class Type, class Iterator>
Iterator find(const Type &key, Iterator first,
        Iterator last)
{
    while (first != last && *first != key)
       ++first;
    return first;
}
```

האלגוריתם הוא **פונקציית תבנית** בעלת שני פרמטרים; הראשון מייצג את סוג האובייקט והשני מייצג את סוג האיטרטור. הפונקציה מקבלת ייחוס לאובייקט שצריך למצוא, ותחום המאופיין על ידי האיטרטורים first ו-last. הפונקציה מחזירה איטרטור לאובייקט הזהה למפתח שנמסר לה, אם נמצא אובייקט כזה, ואחרת היא מחזירה איטרטור הזהה לסוף התחום.

הפונקציה עשויה להחזיר מצביע לסוף האוסף, ולכן האובייקט האחרון בתחום אינו נחשב בתוך התחום. דבר זה מתאפשר הודות לתכונה של C++, שכתובת אחת אחרי המערך היא כתובת חוקית לקריאה, אבל **לא** לכתיבה.

נוהל עבודה זה בעזרת אלגוריתמים הביא את המושג **תכנות גנרי (**generic programming**)**, שבו כותבים פונקציות **שאינן** מכירות את מבנה הנתונים עליהן הן פועלות. הפונקציות, שמממשות אלגוריתם, מקבלות איטרטורים לתחילת התחום ולסופו. פונקציה כזו מניחה שיש לאיטרטור אופרטורים (++) המאפשרים לקדם אותו לאלמנט הבא. כמו כן, מניחה פונקציה כזו שיש אופרטור (*) המקבל את האובייקט הנוכחי כשהאופרטור מופעל על האיטרטור. על פי הנחות אלו יכולה הפונקציה לפעול גם על איטרטורים של מחלקות "מכולה" (או אוספים) וגם על מערכים בסיסיים של השפה.

דוגמה אחרת לאלגוריתם כזה, היא אלגוריתם המבצע **היפוך סדר** של אובייקטים במכולה (אוסף). אם האובייקטים מסודרים בסדר עולה, מסדר אותם האלגוריתם בסדר יורד.

```
template <class Iterator>
void reverse(Iterator first, Iterator last)
{
    --last;
    while (first != last) {
```

```
        swap(*first, *last);
        if (++first == last) break;
        --last;
    }
}
```

כפי שראינו קודם לכן, האלגוריתם מקבל תחום שעליו הוא פועל, ובכל שלב של הלולאה הוא מחליף את האובייקטים בקצות התחום וגם מקטין את התחום. כמו במקרה הקודם, גם כאן יכול אלגוריתם כזה לפעול על מערכים בסיסיים של השפה, או על אוספים. כשהוא פועל על מערכים בסיסיים של השפה הוא מקבל מצביעים, וכשהוא פועל על אוספים הוא מקבל את האיטרטור של האוסף.

## 10.5 מתאמים (Adaptors)

**מתאמים** (adaptors) הם אובייקטים שמקבלים אובייקט אחר, ומשתמשים בו כדי לממש פונקציונליות מסוימת. המתאמים אינם נחשבים כאחד מהמרכיבים העיקריים של הספרייה, אבל הם מציגים רעיון תכנות שראוי לציינו כאן.

המתאם עצמו אינו מממש מבנה אלגוריתמים כלשהו, אלא משמש **כמעטפה** של מחלקה אחרת שמממשת את מבנה הנתונים והפונקציונליות הבסיסית. המעטפה מתאמת את הממשק של המחלקה הבסיסית לממשק מבוקש. למשל, אם רוצים לממש מחסנית, עומדות בפנינו מספר אפשרויות. אפשר לממש מחסנית בעזרת מערך, בעזרת רשימה או בעזרת תור כפול. לכן, המחלקה מחסנית יכולה להיות מתאם שמקבל סוג של מחלקה ומממש את הפונקציונליות הדרושה.

```
template <class Container, class T>
class stack {
    Container c;
public:
    stack() {}
    void push(const T &v)
    { c.push_front(v); }
    T pop()
    { return c.pop_front(); }
};
```

המחלקה stack היא **מעטפה** (envelop) למכולה אחרת, ומשתמשת בה כדי לממש את הפונקציונליות של המחלקה. כל הפונקציות של המחלקה מבוצעות בעזרת המכולה הנתונה לה. כשמגדירים אובייקט ממחלקה זו, מספקים לו את המכולה ואת סוג האובייקטים שבמכולה (ראינו שאפשר להסתדר גם ללא סוג האובייקט). למשל:

```
stack<vector<int>, int> svi;
stack<list<int>, int>   sli;
```

למעשה, הגדרנו מחסנית המבוססת על וקטור, או רשימה. לפי סוג השימוש שלנו במחסנית ניתן לבחור את מבנה הנתונים המתאים יותר.

## 10.6 אובייקטי פונקציות (Function Objects)

**אובייקטי פונקציות** (function objects) אינם נחשבים כאחד החלקים העיקריים של הספרייה, אך ראוי להכיר מונח חשוב זה. אובייקט פונקציה מתנהג בצורה דומה לפונקציה. קיים אופרטור () שאפשר להפעילו כפונקציה. למשל:

```
struct Functor {
    int opreator()()
    { ... }
    ...
};
```

אובייקט זה מעמיס את אופרטור הפונקציה, ולכן אפשר להשתמש באובייקט בצורה הבאה:

```
Functor f;
if (f()) { ... }
```

יכולת זו חשובה כשרוצים להעביר פונקציה, או אובייקט, לאלגוריתמים המשמשים להשוואה בין אובייקטים. האלגוריתם יכול להתייחס לאובייקט כפונקציה, ואז הדרך פתוחה בפני המשתמש להעביר לאלגוריתם פונקציה או אובייקט.

האלגוריתם qsort בספרייה הסטנדרטית של C מקבל פונקציה המאפשרת לקבוע את הסדר הערכי (מספרי או אלפביתי) בין שני אובייקטים, באמצעות ממיון המערך בזיכרון. באופן דומה אפשר לכתוב אלגוריתם המבוסס על פונקציית תבנית ב-C++ למיון מערכים, או מכולות. עלינו לספק לאלגוריתם אובייקט, שמאפשר לקבוע את הסדר בין אובייקטים שונים, או לחילופין, לתת לו פונקציה שמאפשרת זאת.

```
template <class Iterator, class Comp>
void sort(Iterator start, Iterator end, Comp cmp)
{
    //...
    if (cmp(*start, *end) < 0)
    {
      // start is less than end
      ...
    }
    ...
}
```

לפני השימוש באלגוריתם עלינו להגדיר את האובייקט, או הפונקציה, שמאפשרים ביצוע השוואה בין אובייקטים, ולמסור אותם לפונקציית המיון.

## 10.7 סיכום

בפרק זה הצגנו את **ספריית האובייקטים הסטנדרטית** של ++C. סקרנו את החלקים העיקריים שלה ואת רעיונות התכנון שהינחו את מתכנניה. רעיונות אלה לכשעצמם יכולים לשמש אותנו כשאנו מפתחים תוכנה לצרכינו.

הספרייה הסטנדרטית של ++C מורכבת משלושה חלקים עיקריים: **מכולות** (Containers), **איטרטורים** ו**אלגוריתמים**. בניגוד לגישה מוכוונת האובייקטים שבה יש אובייקטים בלבד, ספרייה זו מציגה גישה חדשה שמתאימה ל-++C. בגישה זו יש אלגוריתמים נפרדים **שאינם** קשורים לאובייקטים. אלגוריתמים אלה פועלים על **תחומים** נתונים. האלגוריתמים משמשים כהפשטה בפני עצמה, כי אין הם מכירים את **מבנה הנתונים** שעליו הם פועלים. האלגוריתם דורש איטרטורים, או מצביעים למבנה נתונים זה, שיאפשרו לו לסרוק את מבנה הנתונים.

אלמנטים אחרים המרכיבים את הספרייה כוללים **מתאמים** (adaptors) **ואובייקטי פונקציות** (function objects). המתאמים מאפשרים שימוש במחלקה אחרת והתאמתה למבנה נתונים מסוים. המתאם מאפשר גמישות למשתמש, בכך שהוא מאפשר לבחור את המחלקה המתאימה. לאובייקט המשמש כפונקציה יש אופרטור קריאה לפונקציה, ולכן אפשר להעבירו כפונקציה לאלגוריתם שצריך פונקציה המתארת את יחס הסדר בין אובייקטים שעליו האלגוריתם פועל.

## 10.8 מקורות


1. Musser, D. R. and A. A. Stepanov. "Algorithm oriented generic libraries”, Software Practice and Experience, 24(7):623, July 1994.
2. Musser, D.R and A. Saini. STL Tutorial and Refernce Guide: C++ Programing with the Standard Template Library. Addison Wesly, Reading, MA, 1996.

פרק 11

# מבני נתונים סדרתיים ב-STL

בפרק זה נלמד על **מבני נתונים סדרתיים ב-STL**. מבני נתונים סדרתיים כוללים מערך דינמי, רשימה מקושרת ותור כפול. על מבני נתונים אלה פועלים מרבית האלגוריתמים של STL, ואין צורך לשמור על סדר כלשהו בין האובייקטים המוכלים בהם.

מבני נתונים סדרתיים מאפשרים להכניס לתוכם אובייקטים בכל מקום ולשמור על סדר הכנסתם. אובייקטים שמוכנסים בסדר מסוים מתאימים לסדר הסריקה שלהם. אובייקט שהוכנס ראשון, יהיה ראשון כשמגדירים **איטרטור** על מבני הנתונים. אובייקטים שהוכנסו מאוחר יותר, יופיעו מאוחר יותר בסדר האיטרציה. במבני נתונים כאלה סדר ההכנסה קובע את סדר האיטרציה, ויש אפשרות לעבור על האובייקטים במכולה לפי סדר כניסתם, או בניגוד לסדר הכניסה.

סדר האיטרציה יכול להיות הפוך. אז עוברים מהאובייקט שבסוף המכולה לתחילתה, מאובייקט שהוכנס לסוף המכולה לאובייקט שהוכנס לתחילת המכולה. מכאן אנו למדים שיש סדר של אובייקטים במכולה על פי סדר הכנסתם.

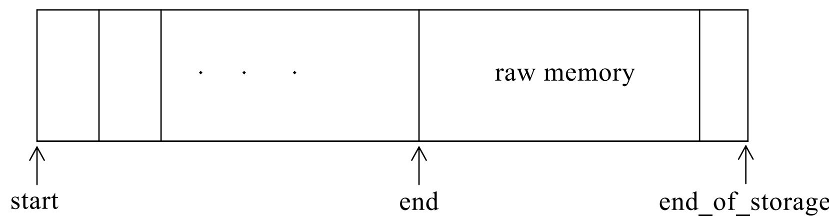
## 11.1 וקטור

מבנה הנתונים **וקטור** מאפשר גישה אקראית לאובייקטים שהוא מכיל. למבנה נתונים זה יש גישה סדרתית, מאובייקט ראשון לבא אחריו בתור, ויש אפשרות לגישה שאינה סדרתית. כלומר, מבנה הנתונים תומך בגישה אקראית על ידי אינדקס, לכל אובייקט בווקטור. הווקטור, כמו רוב המכולות ב-STL, מכיל איטרטור המאפשר לסרוק את האובייקטים שנמצאים בו.

### 11.1.1 מבנה הווקטור

**הגישה האקראית** (random access) הנתמכת על ידי מבנה נתונים זה יעילה (ולכן גם נתמכת) כמו גישה רגילה שנעשית לכל מקום במערך. מבנה הנתונים עצמו מוגדר כמערך של אובייקטים. המערך גדול בדרך כלל ממספר האובייקטים שהוא מכיל, מטעמי יעילות. ברגע נתון יש במערך מספר אובייקטים הנמוך מנפח המערך.

הווקטור הוא **דינמי**, ובעת הצורך הוא גדֵל כדי לאחסן מספר גדול יותר של אובייקטים. הווקטור מכיל שלושה אלמנטים חשובים: מצביע לתחילת אזור הזיכרון של הווקטור, מצביע לסופו ומצביע לאובייקט האחרון שנמצא במערך. כשיש צורך להכניס אובייקט נוסף למערך, נבנה אובייקט חדש בתחום שאינו תפוס עדיין במערך.




***איור 11.1*** *מבנה הווקטור*

הווקטור תומך בפונקציות המאפשרות להכניס אובייקטים בסוף המערך, או בכל מקום בו. פונקציות המכניסות אובייקט באמצע המערך, מעתיקות אובייקטים מהמקום הנדרש לכיוון סוף המערך, כדי להכין מקום לאובייקט החדש. על כן, פונקציות כאלו אינן יעילות. הפונקציה היעילה ביותר היא זו המכניסה אובייקט בסוף אזור האובייקטים (end) ואינה מעתיקה אובייקטים.

אובייקטים המוכנסים למערך אינם זקוקים לבנאי ברירת המחדל. הסיבה לכך נעוצה בעובדה שהמערך אינו נבנה, מלכתחילה, כמערך של אובייקטים, אלא כ**זיכרון רגיל** (raw memory). רק כשמכניסים אובייקט למערך, נבנים בו האובייקטים. גישה זו יעילה יותר גם מהגישה שבה מגדירים את הווקטור באופן הבא:

```
template <class T>
vector {
    T *arr;
    int size;
public:
    vector() : arr(0), size(0) {}
    vector(int sz) { arr = new T[size = sz]; }
    //...
};
```

הבנאי השני של המחלקה וקטור מקצה מערך של אובייקטים. עבור כל אובייקט במערך נקרא בנאי ברירת המחדל. בתחילה הווקטור ריק, ולכן בניית אובייקטים מיותרת. בדרך כלל, קוד המשתמש במערך כזה מכניס אובייקטים למערך מייד לאחר בנייה, וכתוצאה מכך מתבצעת העתקה נוספת של האובייקט המוכנס למערך. על כן, מייד לאחר הכנסתו, מופעל **אופרטור ההעתקה** (copy operator) של האובייקט.

## 11.1.2 האיטרטור

הווקטור מכיל **איטרטור**, ובמקרה זה האיטרטור הוא מצביע לסוג האלמנט המוכל בווקטור. דבר זה מאפשר גישה מהירה לאלמנטים בווקטור ללא כל בזבוז, בדומה למצביעים רגילים בשפה. במילים אחרות, האיטרטור של מכולה זו הוא מצביע, ויש לו כל היתרונות והחסרונות של מצביעים ב-C, או ב-C++.

איטרטור של ווקטור הוא **איטרטור לגישה אקראית** (random access iterator). הוא מאפשר לנוע קדימה או אחורה בווקטור, וגם גישה אקראית לאובייקטים לפי אינדקס. כל האלגוריתמים הקיימים ב-STL פועלים על איטרטור זה.

למחלקה וקטור יש מספר סוגי איטרטורים. לכל אחד מהם פונקציונליות שונה, אבל כולם בנויים על אותו רעיון. סוגי איטרטורים אלה קיימים גם עבור המחלקות האחרות עליהן נלמד בפרק שלפנינו. ההסבר תקֵף גם למחלקות אחרות, ולא נחזור עליו שנית.

איטרטורים מסוג קבוע מאפשרים לסרוק מכולה נתונה, אך אינם מאפשרים לשנות את תכולתה. איטרטורים כאלה שקולים למצביע לקבוע כגון:

```
const char *iter;
```

המצביע אינו מאפשר לשנות את התו שאליו הוא מצביע.

איטרטורים מסוג reverse מאפשרים לסרוק את המכולה בסדר הפוך מסדר האיטרטור הרגיל. פעולת האופרטור "++" מדמה את האיטרציה לאחור במקום קדימה. בדרך זו, מתחילה האיטרציה מסוף המכולה ונגמרת בתחילתו. כשמשתמשים באיטרטור הפוך לווקטור מקבלים אובייקט, ולא מצביע בסיסי, ולכן במקרה זה יורדת היעילות. מומלץ להשתמש באיטרטור הרגיל ולהפעיל את האופרטור "--".

## 11.1.3 פונקציות חשובות

בסעיף זה נסקור פונקציות חשובות של הווקטור, כאלה שהשימוש בהן רב. בהמשך, נראה שימוש בפונקציות אלה. החתימה (prototype) של הפונקציות היא:

```
vector();
iterator begin();
iterator end();
void push_back(const T &val);
void insert(iterator pos, const T &val);
void erase(iterator pos);
T pop_back();
```

הפונקציות המאפשרות איטרציה בווקטור הן הפונקציות begin ו-end. הפונקציה begin מחזירה איטרטור לאובייקט הראשון שבווקטור, והפונקציה end מחזירה איטרטור לכתובת אחת לאחר סוף הווקטור. דבר זה נובע מכך שבשפת C מוגדרת כתובת אחת לאחר סוף המערך ככתובת חוקית, אך אין לבצע על התוכן שלה כל פעולות כלשהן, אלא רק להשוות.

שתי הפונקציות begin ו-end מאפשרות לפתח אלגוריתמים בדרך הבאה:

```
template <class Iter>
void f(Iter start, Iter end)
{
    while (start != end) {
       // ... do what is needed
    }
}
```

בנאי ברירת המחדל (default constructor) של המחלקה וקטור יוצר וקטור ריק שאינו מכיל אובייקטים. כדי להכניס אובייקטים לווקטור נעזרים במספר פונקציות שימושיות (נראה בהמשך). הכנסת אובייקטים לווקטור יכולה לגרום להגדלתו. אם אין מספיק מקום בווקטור, מכפילה המחלקה את אזור הזיכרון שלה. המחלקה מתחילה עם אזור זיכרון אפס, או קבוע מראש, ומכפילה את אזור הזיכרון שבו מאוחסנים האובייקטים. אלגוריתם זה מקטין את מספר הקצאות הזיכרון, ועקב כך את הזמן המבוזבז כתוצאה מכך. לעומת זאת, קיימת אפשרות לבזבוז גדול של זיכרון בהקצאה האחרונה.

הפונקציה **push_back** מכניסה אובייקט נוסף בסוף הווקטור. היא בונה אובייקט הזהה לאובייקט שהיא מקבלת בסוף הווקטור, תוך שימוש בבנאי ההעתקה של האובייקט. אם אין מספיק מקום לאובייקט זה, מוכפל גודל הווקטור.

הפונקציה **insert** מכניסה אובייקט במקום נתון כלשהו לווקטור. גם במקרה זה ייתכן מצב בו יגדל הווקטור. בניגוד לפונקציה הקודמת, ייתכן מצב בו מעתיקה הפונקציה חלק מהאובייקטים בווקטור כדי ליצור מקום לאובייקט החדש. בדרך כלל, פונקציה זו צורכת יותר זמן מהקודמת.

```
void insert(iterator pos, const T &val)
```

האיטרטור מסמן את המקום להכנסת האובייקט, והערך המוכנס הוא הארגומנט השני של הפונקציה.

הפונקציה **erase** מאפשרת ביטול של אובייקט במקום נתון (על ידי איטרטור). פונקציה זו יכולה לגרום להעתקה של חלק מהאובייקטים בווקטור לכיוון תחילתו, כלומר, לכווץ את הווקטור. הפונקציה מפרקת את האובייקט האחרון במערך ומפעילה את המפרק של אובייקט זה.

פונקציה אחרת המבטלת אובייקטים בווקטור היא pop_back. פונקציה זו מבטלת את האובייקט האחרון בווקטור. היא גם מפרקת את האובייקט האחרון על ידי הפעלת המפרק שלו. פונקציה זו אינה מעתיקה אובייקטים, ולכן היא יעילה יותר.

לווקטור קיים אופרטור [] המחזיר ייחוס לאובייקט בווקטור. האופרטור אינו מבצע בדיקת חוקיות על האינדקס, ולכן שתי השגיאות הטיפוסיות הבאות עלולות להתרחש:

- האינדקס המבוקש יכול להיות מחוץ לתחום.
- למרות שהאינדקס בתחום, המקום הנוכחי יכול להיות זיכרון בסיסי, ולא אובייקט.

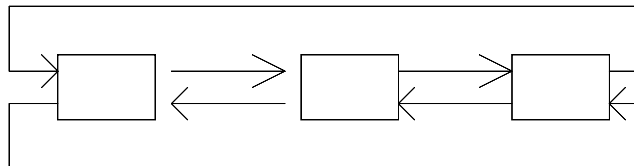
השגיאה השנייה נובעת מכך שמוקצה זיכרון בסיסי (ולא מערך של אובייקטים) והאובייקטים נבנים רק כשיש צורך. מוקצה זיכרון הכפול מגודלו הקודם בפעולת push_back, ורק אובייקט אחד הוכנס לתוכו. שאר האובייקטים רק מציינים מקום ואינם אובייקט. לאובייקטים פשוטים שאין להם פונקציות וירטואליות, וגם למחלקות הבסיסיות שלהן אין פונקציות וירטואליות, אין בעיה. לאובייקטים המכילים פונקציות וירטואליות ההתנהגות אינה מוגדרת.

## 11.2 רשימה כפולה - list

המחלקה **רשימה** (list) היא רשימה מעגלית כפולה, שבכל צומת (איבר) שלה נשמר נתון מתאים. הרשימה אינה שומרת את הצמתים בצורה רציפה בזיכרון, ולכן הכנסות וביטולים של אובייקטים באמצע הרשימה אינם גורמים להעתקת אובייקטים. כמו לכל המכולות, גם בספרייה הסטנדרטית יש איטרטור. האיטרטור מאפשר מעבר קדימה, או אחורה על הרשימה.

### 11.2.1 מבנה הרשימה

הרשימה מכילה צומת אחד המשמש כמצביע לראש הרשימה ולסופה. הרשימה היא מעגלית והמבנה שלה דומה למבנה הרשימה שלמדנו בפרקים קודמים בספר זה. כל צומת ברשימה מכיל מצביע לפנים ולאחור. בנוסף, מחזיק כל צומת ברשימה אובייקט הזהה בסוגו לפרמטר התבנית, כלומר, סוג האובייקטים שברשימה.



***איור 11.2*** *מבנה הרשימה*

הרשימה מאפשרת הכנסת אובייקטים משני צידי הרשימה, וגם הוצאתם משני צידי הרשימה. אפשר לקבל אותו אפקט מווקטור על ידי שימוש בפונקציה erase עם איטרטור לתחילת הווקטור. אך במקרה של וקטור, תהיה הפעולה בזבזנית מאוד ובלתי יעילה. לכן, לא סיפקו מתכנני הספרייה פונקציונליות זו במחלקה וקטור.

### 11.2.2 האיטרטור

איטרטור הרשימה הוא מחלקה מקוננת בתוך הרשימה. הוא מאפשר סריקת הרשימה בצורה דו-כיוונית, ולכן הוא מטיפוס **איטרטור דו-כיווני** (bidirectional iterator). האיטרטור תומך באופרטורים "++" ואופרטורים "--" שמקדמים את האיטרציה לפנים ולאחור, בהתאמה. האיטרטור אינו תומך בגישה אקראית עם האופרטור [].

כדי לאפשר לאלגוריתמים רבים לפעול על הרשימה באמצעות איטרטורים, תומך האיטרטור באופרטורים הבאים:

- `*` מאפשר לקבל את הייחוס לאובייקט הנוכחי שעליו מצביע האיטרטור.
- `==` מאפשר להשוות בין איטרטורים זהים, אם הם מצביעים לאותו צומת.
- `!=` אופרטור השוני מחזיר ערך אמת אם האיטרטורים מצביעים לצמתים שונים.

אופרטורים אלה מאפשרים להתייחס לאיטרטור כאל מצביע בסיסי של השפה, ולכן האלגוריתמים הגנרים של הספרייה יכולים לפעול על רשימות. לעומת האיטרטור של הווקטור, יעילות איטרטור זה נמוכה יותר.

## 11.2.3 פונקציות שימושיות

בסעיף זה נסקור פונקציות שימושיות של הספרייה, אך יש פונקציות נוספות שאינן מתוארות כאן. הקורא מופנה לספרות המקור של הספרייה, כדי לקבל את המידע על כל הפונקציות. הפונקציות שנסקור בפרק זה:

- list();
- iterator begin();
- iterator end();
- void push_front(const T &val);
- void push_back(const T &val);
- void insert(iterator pos, const T &val);
- void erase(iterator pos);
- T pop_front();
- T pop_back();
- void unique();
- void sort();

בנאי ברירת המחדל מגדיר רשימה ריקה, שבה יש צומת אחד בלבד שמצביע אל עצמו. צומת זה הוא **שורש הרשימה** (root) והוא קיים לכל אורך חיי הרשימה.

הפונקציות begin ו-end מחזירות איטרטורים לתחילת וסוף הרשימה, בהתאמה. כשהאיטרטור לסוף הרשימה מצביע, למעשה, לצומת הראשון של הרשימה המשמשת כשורש הרשימה. שתי פונקציות אלו מאפשרות לאלגוריתמים סדרתיים לפעול על הרשימה. נתבונן לדוגמה באלגוריתם הבא:

```
template <class Iter, class Key>
Iter find(Iter start, Iter end, Key k)
{
    while (start != end && *start != k)
       ++start;
    return start;
}
```

```
list<int> li;
//...
list<int>::iterator I = find(li.begin(), li.end(), 1);
if (I != li.end()) {
    // ... found
}
```

הפונקציות push_front ו-push_back מאפשרות הכנסת אובייקטים בתחילת הרשימה ובסופה. הפונקציה insert מוסיפה אובייקט לרשימה במקום הנתון על ידי האיטרטור. משך הזמן לביצוע פונקציות אלו זהה. הפונקציות אינן צריכות להעתיק אובייקטים של הרשימה, גם כשמכניסים אובייקטים באמצע.

הפונקציות pop_front ו-pop_back מאפשרות להוציא אובייקטים מתחילת הרשימה ומסופה, בהתאמה. הפונקציה erase מאפשרת להוציא אובייקט ממקום נתון כלשהו ברשימה. המקום נתון על ידי איטרטור. פונקציות אלו אינן צריכות לצופף את הרשימה, ולכן זמן פעולתן אינו תלוי באורך הרשימה.

הפונקציה sort ממיינת את הרשימה. ב-STL יש פונקציית מיון כללית על פי אלגוריתם quick sort, אך הפונקציה דורשת גישה אקראית לאלמנטים של המכולה שעליה היא פועלת. מכיון שגישה אקראית כזו לא נתמכת על ידי הרשימה, נכתבה פונקציית מיון המתאימה לרשימה.

יש להגדיר אופרטור "<" ו-"==" עבור האובייקטים של הרשימה, מכיון שלרשימה יש פונקציית מיון. כזכור, עבור וקטור לא נזקקנו לאופרטורים אלה.

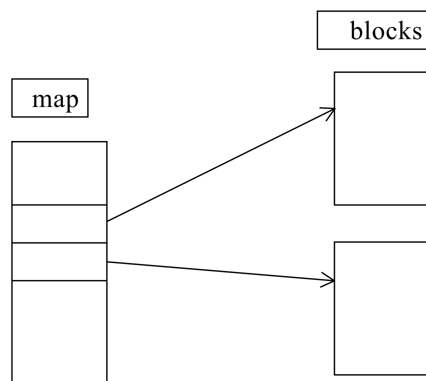
הפונקציה unique מבטלת מופעים כפולים רציפים של אותו אובייקט ברשימה. כלומר, אם יש שני אובייקטים או יותר, הזהים ועוקבים ברשימה, נקבל לאחר הפעלת פונקציה זו אובייקט יחיד. בדרך כלל, פונקציה זו מופעלת לאחר מיון הרשימה. הפעלת הפונקציה unique לאחר מיון מבטלת אובייקטים כפולים ברשימה.

## 11.3 תור כפול

המחלקה **deque** היא **תור כפול**, אשר מאפשר הכנסת אובייקטים משני הקצוות. גם רשימה כפולה יכולה לשמש כתור כפול, אך ב-STL התור הכפול אינו מיושם באמצעות רשימה, אלא בעזרת מערך של מצביעים. כמו לכל מכולה אחרת, גם לתור הכפול יש איטרטור ב-STL.

### 11.3.1 מבנה התור הכפול

התור הכפול מיושם באמצעות מערך מצביעים אל מערכים של אובייקטים. סידור זה מאפשר הכנסת אובייקטים לתחילת התור, או לסופו, ללא צורך בהעתקות. כשצריך להוסיף אובייקט בהתחלה ואין מקום, מוסיפים בלוק של אובייקטים ומוסיפים מצביע כזה בתחילת המפה שמצביעה לבלוקים של אובייקטים.




***איור 11.3** מבנה התור הכפול*

איור 11.3 מתאר תור כפול עם שני בלוקים מוקצים. במפה יש מצביעים לשני הבלוקים. כל בלוק הוא בעל גודל קבוע. כשיוצרים אובייקט כזה, אין מפה ואין בלוקים (מערכים) של אובייקטים. כשמנסים להכניס אובייקט ראשון, מוקצה הבלוק הראשון והמפה מאותחלת, כשהמצביעים של התחלת המפה וסופה מאותחלים לאמצע הבלוק. מצביעים אלה הם איטרטורים של מחלקה זו. האיטרטורים מאותחלים להצביע לאמצע הבלוק הראשון. בצורה זו אפשר להכניס אובייקטים לשני צידי התור ללא צורך בהעתקות.

כשמנסים להכניס אובייקט לסוף תור (או לתחילתו) והאיטרטור מצביע לבלוק האחרון של המפה שבסוף הבלוק האחרון, יש צורך להקצות בלוק חדש של אובייקטים. יש צורך להגדיל את המפה, כדי שתוכל לקבל את המצביע לבלוק זה. הגדלת המפה מתבצעת על ידי הוספת מצביע חדש. מחיר הוספת מצביע חדש נמוך. דבר זה נובע מכך שבדרך כלל יש מספיק מקום למצביעים נוספים במפה. רק כשאין מספיק מקום במפה, יש להעתיק את מצביעי המפה לאזור זיכרון חדש, וגדול יותר. ברוב המקרים גלישה שכזו לא תהיה ולכן לא תהיינה העתקות כלשהן.

כשמנסים להכניס אובייקט לאמצע תור יש צורך בהעתקה, מכיון שבלוקים הם מערכים רציפים של אובייקטים. התור הכפול יעיל במיוחד כשמכניסים אובייקטים משני קצותיו, והוא יעיל יותר מרשימה, כי אין צורך לטפל במצביעים. לעומת זאת, הכנסת אובייקטים באמצע התור אינה יעילה. הרשימה הכפולה יעילה יותר.

## 11.3.2 האיטרטור

האיטרטור בהקשר זה **אינו** מצביע, אם כי יעילותו מתקרבת ליעילות מצביעים בסיסיים בשפה. האיטרטור הוא מחלקה פנימית במחלקה המייצגת תור כפול. האיטרטור הזה הוא מסוג גישה אקראית, ולכן הוא תומך בהשוואה בין מצביעים (<), בגישה אקראית, חיבור ותוספת. האיטרטור מספק אופרטורים (כגון "=+", "*", "==", "++" ,"--") ובכך הוא מתנהג כמו מצביע רגיל ב-++C.

ברוב המקרים, נעשה קידום האיטרטור באמצע בלוק ולכן, פרט להגדלת מצביע, לא נעשה דבר. במקרים אלה הפעולה יעילה כמעט כמו הגדלת מצביע בסיסי של השפה. עם זאת, האיטרטור נמצא בסוף בלוק ויש לקדמו לבלוק הבא. קטע הקוד הבא לקוח מהספרייה ומדגים את אופן קידום האיטרטור.

```
iterator& operator++() {
    if (++current == last) {
      first = *(++node);
      current = first;
      last = first + buffer_size;
    }
    return *this;
}
```

כשהאיטרטור מגיע אל המצביע האחרון בבלוק, הוא מאותחל לכניסה הבאה במפת הבלוקים. הכניסה הנוכחית במפת הבלוקים מיוצגת על ידי node. לאחר מכן, מעודכן המצביע לסוף הבלוק לפי גודל הבלוק. ברוב המקרים פעולת קידום (או נסיגה) של האיטרטור, יעילה כמעט כמו ביצוע הפעולות המקבילות על מצביעים של C או ++C.

## 11.4 מבנה נתונים - דוגמה

את מבני הנתונים נלמד בעזרת דוגמה העוסקת ביומן טלפונים. דוגמה זו תשמש אותנו בכל מחלקות STL שנלמד בפרק זה. במקום אחד (שנראה בהמשך) נציין את סוג המכולה שנשתמש בה. טכניקה זו מאפשרת לשנות את סוג המכולה ללא שינוי התוכנה המשתמשת בה.

### 11.4.1 מבנה היומן

יומן הטלפונים מורכב ממספר גדול של כניסות. לכל כניסה ביומן יש שם אדם כלשהו ומספר הטלפון שלו. היומן נרשם לקובץ ASCII ונקרא מתוכו, בעת הצורך. כל רשומה ביומן היא שורה בקובץ. רשומה ביומן מיוצגת על ידי אובייקט מהמחלקה DiaryEntry.

```
#include <string.h>
#include <fstream.h>

class DiaryEntry {
public:
   enum { name_len = 32, phone_len = 12 };
private:
   char name[name_len];
   char phone[phone_len];
public:
```

```
    DiaryEntry()
    { name[0] = 0;
      phone[0] = 0; }
      void set_name(const char *nm)
      { if (nm) strncpy(name, nm, name_len);
        else name[0] = 0; }
      void set_phone(const char *pn)
      { if (pn) strncpy(phone, pn, phone_len);
        else phone[0] = 0; }
      //
      // get functions
      //
      const char *get_name() const
      { return name; }
      const char *get_phone() const
      { return phone; }
};
```

בנאי הרשומה ביומן מאפס את השדות שם וטלפון, ומספק פונקציות השמה לשם ולמספר הטלפון ופונקציות גישה לשדות אלה. הקוד של מחלקות אלו ושל הפונקציות הגנריות של היומן נמצא בקבצים **diary.cpp** ו-**diary.h**.

## 11.4.2 פונקציות עזר

ליומן יש מספר פונקציות תבנית המשמשות לקריאה וכתיבה בקבצים. פונקציות אלו קוראות, או כותבות מכולה שבה אובייקטים מסוג DiaryEntry.

כדי לפלוט, או לקלוט אובייקט מסוג DiaryEntry מוגדרים האופרטורים "<<" ו-">>". האופרטור "<<" מקבל ייחוס לאובייקט מסוג ostream, ומחזיר ייחוס לאובייקט כזה. מכיון שהאופרטור מקבל ייחוס לאובייקט מסוג ostream, הוא יכול לפעול על כל אובייקט שיורש מ-ostream. האופרטור "<<" מקבל ייחוס לאובייקט מסוג DairyEntry, שאותו הוא מדפיס על זרם הפלט הנתון לו. באופן דומה מוגדר האופרטור ">>" עבור זרמי קלט.

```
ostream &operator<<(ostream &os, const DiaryEntry &de);

istream &operator>>(istream &is, DiaryEntry &de);
```

פונקציית התבנית read מאפשרת לקרוא קובץ המכיל כניסות של יומן הטלפונים לתוך מכולה כלשהו. המכולה הנתון לפונקציה הוא פרמטר של פונקציית התבנית read. פונקציה זו מאפשרת לקרוא את הקובץ לכל מכולה, כשהפונקציה מניחה שהמכולה היא בעלת פונקציה push_back, המוסיפה אובייקט בסופה. לצורך קריאת אובייקט, משתמשת הפונקציה באופרטור ">>". הפונקציה מחזירה את מספר האובייקטים שנקראו מהקובץ.

```
// read into a given container and returns the number
// of objects read
template <class Container>
int read(Container &c, const char *file)
{
    int count = 0;
    ifstream in(file);
    while (in) {
      DiaryEntry de;
      in >> de;
      c.push_back(de);
      ++count;
    }
    return count;
}
```

פונקציית התבנית apply מקבלת שני פרמטרי תבנית. הראשון, מכולה (שמכילה אובייקטים) והשני, פונקציה שיש להפעיל על כל האובייקטים במכולה. הפונקציה סורקת את המכולה ומפעילה את הפונקציה f על כל אחד מהאובייקטים. הפונקציה מקבלת ייחוס למכולה, ולכן היא אינה פועלת עבור מערכים בסיסיים של השפה. אילו רצינו להפעיל את הפונקציה עבור מערכים בסיסיים, היינו צריכים לקבל תחום (begin, end).

```
template <class Container, class Func>
int apply(Container &c, Func f)
{
    Container::iterator start(c.begin()), end(c.end());
    int count = 0;
    while (start != end)  {
      ++count;
      f(*start++);
    }
    return count;
}
```

כדי להדפיס אובייקטים לפלט, נשלב מספר אובייקטים ואובייקטי פונקציות. הראשון שבהם, printer, הוא אובייקט-פונקציה. כלומר, זהו אובייקט המתפקד כפונקציה, מכיל ייחוס לזרם פלט ומדפיס לזרם הנתון אובייקטים הניתנים לאופרטור הפונקציה שלו.

```
struct printer {
    ostream &os;
    printer(ostream &o) : os(o) {}
    void operator()(const DiaryEntry &v)
    { os << v; }
};
```

כדי לכתוב אובייקט מסוג DiaryEntry נשתמש בשתי פונקציות התבנית write. הראשונה מביניהן מקבלת מכולה שבה אובייקטים להדפסה ואובייקט מסוג זרם פלט (out). הפונקציה מגדירה אובייקט פונקציה prt, המתייחס לזרם הפלט. המכולה מודפסת בעזרת הפונקציה apply.

פונקציית ההדפסה השנייה מקבלת מכולה שבה אובייקטים להדפסה ושם של קובץ. הפונקציה יוצרת אובייקט out, מסוג זרם פלט. בנוסף, מגדירה הפונקציה אובייקט-פונקציה ומשתמשת בפונקציה apply כדי להדפיס את המכולה הנתון לה.

שתי פונקציות ההדפסה write יכולות להדפיס מכולה המכיל אובייקטים שונים, לאו דווקא DiaryEntry, כל עוד יש לאובייקטים אלה אופרטור הדפסה מתאים (<<). הטכניקה המודגמת כאן היא טכניקה גנרית, ולכן חשוב להכיר אותה, כדי לנצל את יכולתה .

```
template <class Container>
int write(Container &c, ostream &out)
{
    int count = 0;
    if (out) {
      printer prt(out);
      count = apply(c, prt);
    }
    return count;
}
// write a given container into a given file
template <class Container>
int write(Container &c, const char *file)
{
    ofstream out(file);
    if (out) {
      printer prt(out);
      return apply(c, prt);
    }
      return 0;
}
```

הפונקציה find היא פונקציה גנרית המוצאת את האובייקט הראשון בתחום נתון על פי קריטריון מסוים. התחום נתון על ידי שני איטרטורים (או מצביעים) והקריטריון נתון על ידי אובייקט-פונקציה. אם קיים אובייקט בתחום הנתון לפונקציה find, שאליו אובייקט הפונקציה מחזיר ערך אמת (שונה מאפס) יוחזר איטרטור (או מצביע) לאובייקט זה, אחרת יוחזר איטרטור לסוף התחום.

```
// find searches a container and returns a pointer to the
// iterator position where the requested object. If no
// found returns an iterator one past the end of the
// container
```

```
template  <class Iterator, class Func>
Iterator find(Iterator start, Iterator end, Func f)
{
   while (start != end)
      if (f(*start))
            return start;
      else
            ++start;
   return start;
}
```

האופרטורים של הקלט והפלט משתמשים בפונקציות הציבוריות של DiaryEntry כדי לקרוא, או לכתוב, אובייקט מסוג זה לזרם קלט או פלט נתון.

```
ostream &operator<<(ostream &os, const DiaryEntry &de)
{
   return os << de.get_name()
              << "\t\t\t" << de.get_phone() << endl;
}

istream &operator>>(istream &is, DiaryEntry &de)
{
   char buf[DiaryEntry::name_len];
   is >> buf;
   de.set_name(buf);
   is >> buf;
   de.set_phone(buf);
   return is;
}
```

אופרטור הקלט מסובך יותר, משום שעליו לקרוא את השם ומספר הטלפון לתוך חוצצים מקומיים, ורק אז להעבירם לכניסה ביומן.

## 11.4.3 התוכנית הראשית

התוכנית הראשית (**diary_t.cpp**) מכילה את פונקציות הניהול ואת ממשק המשתמש שמקבל פקודות מהמשתמש בתוכנית.

כדי לכלול את הקבצים המתאימים, עלינו לכלול את הקבצים vector, list ו-deque שנמצאים בספרייה STL. התוכנית כוללת את קבצים האלה משום שהיא זקוקה לשלושתם.

```
#include "Diary.H"
#include "stl\vector.h"
#include "stl\list.h"
#include "stl\deque.h"
#include <fstream.h>
```

המשתמש יכול להפעיל פקודות הוספה או ביטול של רשומה ביומן, עדכון רשומה, קריאה או כתיבה של קובץ המכיל את יומן הטלפונים. הוא יכול להציג את היומן הנוכחי וגם לצאת מהתוכנית. אפשרויות אלו מפורטות בפקודה enum הבאה:

```
enum { add_entry_cmd, remove_entry_cmd, update_entry_cmd,
    read_file_cmd, write_file_cmd, display_cmd, exit_cmd };
```

כדי להגדיר את סוג המכולה (container) שבו משתמשת התוכנית במקום אחד בלבד, נשתמש ב- typedef. אם נרצה לשנות את סוג המכולה עלינו להכניס להערה שתי שורות מהשורות הבאות, ולהוציא מההערה את המכולה המבוקש. אין צורך לתקן במקומות אחרים משום שבכל מקום אחר אנו משתמשים בסוג diary_container.

```
//typedef vector<DiaryEntry> diary_container;
//typedef list<DiaryEntry> diary_container;
typedef deque<DiaryEntry> diary_container;
```

האופרטורים "<" ו-"==" משמשים להגדרת יחס סדר של אובייקטים מסוג DairyEntry, כדי לאפשר מיון מכולה שבה אובייקטים מסוג זה. פונקציית המיון ו-unique שייכות למחלקה list, ולכן הן אינן אלגוריתמים גנריים ויש לספק להן את אופרטורי הסדר האלה:

```
// the following operators are for the list functions merge
// and unique to work
int operator<(const DiaryEntry &d1, const DiaryEntry &d2)
{
      return (strcmp(d1.get_name(), d2.get_name()) < 0);
}
int operator==(const DiaryEntry &d1, const DiaryEntry &d2)
{
      return (strcmp(d1.get_name(), d2.get_name()) == 0);
}
```

האובייקט equal_func הוא **אובייקט-פונקציה** (function object) המכיל ייחוס לאובייקט מסוג DiaryEntry. אובייקט הפונקציה מגדיר את אופרטור הפונקציה ומחזיר ערך אמת כשהוא מקבל כניסה ביומן, שהיא בעלת שם זהה לשם של הייחוס de שמכיל אובייקט הפונקציה. כך אפשר להשוות אובייקט כלשהו לאובייקט שאליו מתייחס אובייקט הפונקציה. אובייקט זה משמש לאלגוריתמים כגון find שעוברים על תחום ומוצאים מופע של אובייקט בתוכו.

```
struct equal_func {
    const DiaryEntry &de;
    equal_func(const DiaryEntry &d) : de(d) {}
    int operator()(const DiaryEntry &d)
    { return (strcmp(de.get_name(), d.get_name()) == 0); }
};
```

הפונקציה diaplay_menu מציגה למשתמש תפריט שמכיל את אוסף הפקודות הניתנות לביצוע. הפונקציה מקבלת פקודה מהמשתמש ומחזירה את מספר הפקודה שבחר. התפריט מוצג על המסך (לא בחלונות) כשלכל אפשרות בתפריט יש מספר.

```
int display_menu()
{
    int res = exit_cmd;
    cout << add_entry_cmd << ") add a new entry\n";
    cout << remove_entry_cmd << ") remove an entry\n";
    cout << update_entry_cmd << ") update an entry\n";
    cout << read_file_cmd << ") read from a file\n";
    cout << write_file_cmd << ") write into a file\n";
    cout << display_cmd << ") display the current diary\n";
    cout << exit_cmd << ") exit from program\n";
    cout << "enter: ";
    cin >> res;
    cout << endl;
    return res;
}
```

הפונקציה add_entry מוסיפה כניסה ליומן הטלפונים. הפונקציה מבקשת מהמשתמש להכניס את פרטי הכניסה. היא משתמשת באובייקט מסוג DiaryEntry ובאופרטור "<<" כדי להכניס את המידע לאובייקט. לבסוף, היא מוסיפה את האובייקט למכולה הנתונה לה.

```
void add_entry(diary_container &dc)
{
    DiaryEntry de;
    cout << "enter details:";
    cin >> de;
    dc.push_back(de);
    cout << endl;
}
```

הפונקציה remove_entry מאפשרת למחוק אובייקט מיומן הטלפונים. כמו בפונקציה הקודמת, גם היא מגדירה אובייקט מסוג DiaryEntry וקוראת לתוכו את פרטי האובייקט המיועד לביטול. לאחר מכן, היא מגדירה אובייקט-פונקציה מסוג equal_func ומוצאת את מיקום האובייקט המבוקש בעזרת הפונקציה find. אם אכן נמצא אובייקט כזה במכולה, הוא מבוטל ממנה בעזרת הפונקציה erase. הפונקציה erase של המכולה מקבלת איטרטור למקום שצריך לבטל.

```
void remove_entry(diary_container &dc)
{
    DiaryEntry de;
    cout << "enter details:";
    cin >> de;
```

```
    equal_func eq(de);
    diary_container::iterator pos = find(dc.begin(),
                                         dc.end(), eq);
    if (pos != dc.end())
      dc.erase(pos);
    else
      cout << "not found";
    cout << endl;
}
```

הפונקציה update_entry פועלת בצורה דומה לפונקציה remove_entry. הפרטים החדשים של האובייקט המבוקש נקלטים לאובייקט de מהמשתמש. במקרה זה, אם נמצא האובייקט המבוקש, הוא מעודכן בעזרת אופרטור ההשמה של האובייקט de לאובייקט במכולה. הפונקציה משתמשת באופרטור ההשמה של האובייקט DiaryEntry, כדי לעדכן את האובייקט במכולה.

```
void update_entry(diary_container &dc)
{
    DiaryEntry de;
    cout << "enter details:";
    cin >> de;
    diary_container::iterator i = find(dc.begin(),
                              dc.end(),equal_func(de));
    if (i != dc.end())
            *i = de;
    else
            cout << "not found";
    cout << endl;
}
```

הפונקציה read_file מקבלת מכולה שיש לקרוא לתוכה אובייקטים. את שם הקובץ היא מבקשת מהמשתמש. הפונקציה מבטלת את כל האובייקטים במכולה, ואחר כך קוראת לפונקציה read כדי לקרוא את הקובץ ולמלא את המכולה. ביטול האובייקטים נעשה בעזרת הפונקציה erase. בגירסה השנייה של פונקציה זו (erase) היא מקבלת תחום במכולה המיועד למחיקה. לכן, מועבר לפונקציה התחום של כל המכולה.

```
void read_file(diary_container &dc)
{
    char buf[128];
    cout << "enter file name:";
    cin >> buf;
    cout << endl;
    dc.erase(dc.begin(), dc.end());
    read(dc, buf);
}
```

הפונקציה write_file מקבלת מכולה של יומן ומדפיסה לקובץ את האובייקטים שנמצאים בה. היא מבקשת את שם הקובץ מהמשתמש ומכניסה אותו לחוצץ מקומי. הפונקציה משתמשת בפונקציה write כדי לכתוב לקובץ את המכולה הנתון.

```
void write_file(diary_container &dc)
{
    char buf[128];
    cout << "enter file name:";
    cin >> buf;
    cout << endl;
    write(dc, buf);
}
```

הפונקציה display מציגה לפלט הסטנדרטי את האובייקטים שבמכולה יומן. היא משתמשת באופרטור "<<" הפועל על כניסה ביומן ובזרם הפלט כדי להדפיס את היומן לפלט הסטנדרטי.

```
void display(diary_container &dc)
{
    diary_container::iterator start = dc.begin(),
                               end = dc.end();
    while (start != end)
      cout << *start++;
}
```

הפונקציה הראשית של הדוגמה מגדירה את המכולה שבה אובייקטים מסוג DairyEntry. היא מבצעת לולאה כל עוד אינה מקבלת מהפונקציה diaply_menu ערך השונה מ-exit_cmd. על פי הערך המוחזר מתבצעת הפונקציה הדרושה מתוך אפשרויות case.

```
int main()
{
    int cmd;
    diary_container diary;
    while ((cmd = display_menu()) != exit_cmd) {
        switch (cmd) {
        case add_entry_cmd:
             add_entry(diary);
             break;
        case remove_entry_cmd:
             remove_entry(diary);
             break;
        case update_entry_cmd:
             update_entry(diary);
             break;
        case read_file_cmd:
```

```
            read_file(diary);
            break;
        case write_file_cmd:
            write_file(diary);
            break;
        case display_cmd:
            display(diary);
            break;
        case exit_cmd:
        default:
            break;
        }
    }
    return 0;
}
```

## 11.4.4 תכנות גנרי

**STL** קידמה מאוד את נושא **התכנות הגנרי**. בתכנות גנרי (generic programming) מוסיפים פונקציות מופשטות המטפלות במספר רב של מבני נתונים, ללא כל ידיעה מראש על סוג מבני הנתונים. היתרון בכך הוא שאין צורך לכתוב שוב את אותה פונקציה עבור מבנה נתונים מסוים, או אחר.

לדוגמה, פונקציה כזו היא find. הפונקציה מקבלת תחום שעליו היא פועלת. תחום זה מאופיין על ידי שני איטרטורים המתנהגים כמצביעים. הם מאפשרים לסרוק את התחום ולקבל בעזרת האופרטור "*" את הערך הנוכחי של האובייקט המוצבע על ידי האיטרטור. בצורה דומה ניתן להגדיר את הפונקציות apply ו-write.

הפונקציה generic_apply מקבלת תחום שעליו היא פועלת. התחום יכול להיות נתון על ידי מצביעים בסיסיים של השפה, או על ידי איטרטורים התומכים במספר פעולות בסיסיות. הפעולות הבסיסיות יכולת לקדם את האיטרטור על ידי האופרטור "++", או לקבל את תוכנו של האיטרטור על ידי האופרטור "*". הדרישה האחרונה מהאיטרטור היא יכולת ההשוואה בין איטרטורים.

הפונקציה מקבלת פרמטר של תבנית שיכול להיות מצביע לפונקציה, או אובייקט-פונקציה (function-object). הפונקציה מפעילה את האחרון (הפונקציה, או אובייקט הפונקציה) על התחום הנתון לה.

```
// the following is a generic apply function which works on
// a given range. Therefore the function is not bound to a
// container,it can also work on a C++ builtin array. The
// function returns the number of time it applied its given
// functor f
template <class Iter, class Func>
int generic_apply(Iter start, Iter end, Func f)
```

```
{
    int count = 0;
    while (start != end) {
       ++count;
       f(*start++);
    }
    return count;
}
```

הפונקציה generic_write היא פונקציה גנרית המאפשרת לכתוב תחום נתון לכל פלט. כמו במקרה הקודם, גם פונקציה זו פועלת על תחום המאופיין על ידי איטרטורים, או מצביעים. האיטרטורים או מצביעים, חייבים לתמוך בתכונות שתארנו כבר.

הפונקציה מקבלת פרמטר נוסף של תבנית, המייצג את הפלט, אשר אינו חייב להיות קובץ, אלא למשל מחלקה אחרת שתומכת באופרטור "<<". העלינו את רמת ההפשטה של הפלט!

למשל, אם נגדיר את האופרטור "<<" שיכניס אובייקטים לזרם נתון עבור מכולה כלשהי, נוכל להשתמש בו כדי להעתיק אובייקטים מהתחום הנתון למכולה הנתונה.

```
// the following is a generic write function which works on
// a given range and prints it out. It uses the genric
// programming approach to work on builtin arrays or
// containers of STL
template <class Iter, class Output>
void generic_write(Iter start, Iter end, Output &out)
{
    while (start != end)
       out << *start++ << ' ';
    out << endl;
}

template <class Container, class T>
class adaptor {
private:
      Container &c;
public:
      adaptor(Container &cont) : c(cont) {}
      adaptor &operator<<(const T &v)
      { c.push_back(v); return *this; }
      //...
};
```

המחלקה adaptor היא מחלקת תבנית המאפשרת להכניס אובייקטים למכולה כלשהי. מחלקה זו יכולה לשמש לצורך העתקת תחומים לתוך מכולות. את מימוש המחלקה adaptor אני משאיר כתרגיל לקורא.

## 11.5 סיכום

בפרק זה סקרנו מספר מחלקות של הספרייה הסטנדרטית **STL**. אלו הן מכולות שבהן נמצאים אובייקטים שונים. למחלקות אלו יש תכונות ומבנה נתונים שונים ולכן הן עשויות להתאים במידה שונה לסוגי יישומים שונים.

המחלקה **vector** היא מחלקת תבנית המייצגת וקטור דינמי. וקטור זה גדל, בעת הצורך. הווקטור דומה למערכים הסטנדרטיים של ++C, אשר האובייקטים שנמצאים בו נשמרים במקום רציף בזיכרון. כתוצאה מכך:

- הכנסת אובייקטים בסוף המערך יעילה מאוד, ואילו הכנסת אובייקטים באמצע המערך אינה יעילה כלל.
- סריקת הווקטור בעזרת איטרטור יעיל מאוד.

המחלקה **list** היא רשימה מקושרת מעגלית כפולה. מחלקה זו אינה שומרת את האובייקטים בצורה רציפה, ולכן הכנסת אובייקטים לאמצע המחלקה יעילה באותה מידה של הכנסה בקצוות. הכנסת אובייקטים בסוף הרשימה יעילה פחות מהפעולה הדומה בווקטור. סריקת רשימה כזו יעילה פחות מסריקת וקטור.

המחלקה **deque** היא תור המאפשר להכניס אובייקטים בשני צידיו. כלומר, אפשר להכניס אובייקטים לתחילת התור, או לסופו. התור הכפול מורכב מבלוקים שבהם אובייקטים שמאפשרים להכניס אובייקטים בתחילה, או בסוף התור, בקלות יחסית. גם במקרה זה, הכנסת אובייקט באמצע התור גורמת להעתקה של אובייקטים ולכן התור פחות יעיל מרשימה בנקודה זו.

ראינו את סגנון **התכנות הגנרי** (generic programming) שפועל על תחום נתון. סגנון זה מאפשר לכתוב אלגוריתמים גנרים על מכולהים של STL, או על מבני נתונים בסיסיים של ++C.

## 11.6 שאלות

1. השלם את המחלקה adaptor כך שניתן יהיה להכניס אלמנטים גם למכולות של STL. עליך להתייחס גם להכנסת תווים ו-endl.
2. ממש פונקציה גנרית המאפשרת לקרוא מקובץ לתוך מערך או מכולה.
3. ממש רשימה גנרית מקושרת (יחידה) בסגנון STL.
4. ממש איטרטור המאפשר לכתוב **לתוך** זרם פלט.
5. ממש איטרטור המאפשר לכתוב **מ**זרם פלט.
6. ממש אלגוריתם מיון עבור הרשימה, הפועל בשיטת מיון ממוזג. בשיטה זו מחלקים את תחום המיון לשני חלקים, ממיינים כל חלק בנפרד ואז ממזגים את שני החלקים. מה הסיבוכיות (זמן ומקום) של אלגוריתם זה?

## פרק 12

# מבנים שאינם סדרתיים ב-STL

בפרק זה נלמד את המבנים בספרייה הסטנדרטית STL שאינם סדרתיים. מכולות אלו אינן סדרתיות משום שסדר הכנסת האובייקטים אליהן אינו נשמר, ובעת איטרציה על המכולה, כלומר סריקה, אין כל קשר בין סדר הכנסת האובייקטים לסדר האיטרציה.

הדבר האופייני למבנים אלה הוא מהירות החיפוש הגבוהה של אובייקטים במכולות. הדבר חשוב במיוחד למבנה נתונים, כמו קבוצה,אשר מחייב מהירות חיפוש גבוהה.

כל המכולות המוצגות בפרק שלפנינו מתבססות על מבנה הנתונים **עץ אדום-שחור** (Red Black Tree). עצים אלה הם **עצי חיפוש בינאריים מאוזנים** (binary balanced search tree). לכל צומת יש מצביעים לתת-עץ שמאלי ולתת-עץ ימני, וגם מצביע לצומת האב. לכל צומת יש גם ציון צבע אדום או שחור, ולכן עץ מסוג זה נקרא **עץ אדום-שחור**.

בפרק זה נלמד אודות המכולות האלו : **set**, **multiset**, **map** ו-**multimap**. כל אחת מהן מייצגת אוסף, המאפשר פעולות חיפוש מהירות.

## 12.1 הגדרת קבוצות

המונח **קבוצה** (set) מגדיר אוסף של אובייקטים ללא חזרות. ההגדרה המתמטית של קבוצה אינה מתייחסת לסדר בין האובייקטים השונים בקבוצה. קבוצה יכולה להיות אוסף סופי, או אינסופי. נתרכז באוספים סופיים, אשר יכולים להיות גדולים מאוד. העובדה שקיימים אוספים כאלה (קבוצות) לכשעצמה אינה אומרת דבר, ולכן מוגדרות פעולות מסוימות על קבוצות.

## 12.1.1 פעולות על קבוצות

את הפעולות המוגדרות על קבוצות אפשר לחלק לשני סוגים :

- פעולות המשנות את הקבוצה.
- פעולות שאינן משנות את הקבוצה.

פעולות המשנות את הקבוצה מכניסות אליה, או מוציאות ממנה אובייקטים. פעולות אלו, בהכרח, משנות את מספר האובייקטים בקבוצה. פעולת insert שמכניסה אובייקט לקבוצה, היא דוגמה לסוג הראשון. בתחילת ביצוע ההכנסה יש לבדוק אם האובייקט קיים בקבוצה ואז, אם אינו קיים - להכניסו. לביצוע פעולה זו דרוש חיפוש של האובייקט בתוך הקבוצה. מכאן שדרושה לנו פעולת חיפוש מהירה.

פעולה אחרת המשנה את הקבוצה היא הפעולה המבטלת אובייקט מהקבוצה. גם במקרה זה יש למצוא את האובייקט המבוקש, ואז להרחיק אותו ממנה. שוב עולה הצורך בפעולת חיפוש מהירה בקבוצה.

פעולות אחרות שימושיות על קבוצות הן פעולת החיתוך ופעולת האיחוד. בפעולת החיתוך יוצרים קבוצה משתי קבוצות נתונות. קבוצת התוצאה מכילה את כל האובייקטים המשותפים (מופיעים) בשתי קבוצות הקלט של פעולה זו. כמובן שגם צריך לחפש אובייקטים בקבוצה אחת ולסרוק את כל האובייקטים בקבוצה האחרת. גם כאן יש צורך בפעולת חיפוש מהירה.

בפעולת האיחוד מתקבלות שתי קבוצות שמהן בונים קבוצה שלישית המייצגת את איחוד שתי הקבוצות הנתונות. קבוצת האיחוד מכילה את האיברים של שתי קבוצות הקלט, כלומר, כל איבר שמופיע בקבוצה הראשונה או בשנייה, יופיע בקבוצת האיחוד.

בפעולות שתוארו עד כה ראינו מדוע מהירות החיפוש חשובה, במיוחד לקבוצות. כדי לממש קבוצה בצורה יעילה, עלינו לספק מבנה נתונים שמאפשר חיפוש מהיר.

## 12.1.2 מבני נתונים אפשריים לקבוצה

כדי לספק מנגנון חיפוש מהיר של אובייקטים, איננו יכולים להסתמך על מבנה נתונים סדרתי, כמו רשימה. חיפוש אובייקט ברשימה שאינה מסודרת יהיה יחסי למספר האובייקטים שנמצאים בה. למשל, אם קיימים 100,000 אובייקטים ברשימה, נצטרך לסרוק על 50,000 אובייקטים בממוצע, כדי למצוא אובייקט נתון.

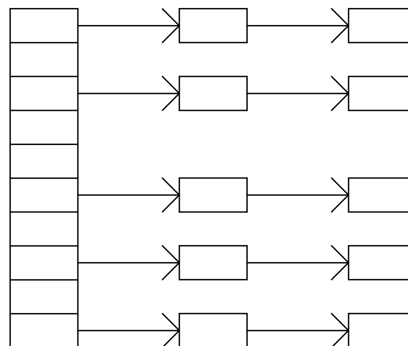
מימוש קבוצה צריך, אם כן, להיעשות על ידי מבנה נתונים שזמן החיפוש בו אינו לינארי. יש מספר מבני נתונים שאינם לינאריים. בהמשך נלמד לפחות שניים כאלה.

### 12.1.2.1 טבלת ערבול

אחד ממבני הנתונים שיכולים לשמש ליצירת קבוצה הוא **טבלת ערבול** (Hash Table). טבלה כזו מאפשרת למצוא אובייקט בזמן קצר מאוד. כדי לא לחפש אובייקט לאורך

רשימה אחת, מחלקים את מרחב החיפוש למספר גדול של רשימות. השאיפה היא לחלק את כל האובייקטים בין כל הרשימות באופן שווה, כך שבכל רשימה ייִמָּצא מספר אובייקטים קטן יחסית. את הרשימות מחזיקים במערך של רשימות. לכל אובייקט יש **ערך ערבול** (Hash Value) שבעזרתו קובעים את הרשימה שבה עשוי להימצא האובייקט.

**מבנה טבלת ערבול:**



בטבלה המתוארת יש עשרה אובייקטים, אך חיפוש אובייקט בה הוא בסדר גודל של השוואה אחת, ולא חמש. הסיבה לכך היא, שאת הרשימה המתאימה לערך מבוקש אנו מקבלים מערך הערבול בפעולה מיידית. כלומר, בעזרת ערך הערבול מקבלים את האינדקס של הרשימה המתאימה במערך הרשימות. חישוב ערך הערבול אינו תלוי במספר האובייקטים הכללי בטבלה. כעת שקולה מציאת האובייקט לסריקה של רשימה קצרה, שני אובייקטים במקרה זה.

כדי לממש טבלת ערבול, יש לספק **פונקציית ערבול** (Hash Function) המקבלת אובייקט ומוצאת את הכניסה המתאימה לה במערך. בנוסף, יש לקבל **פונקציית השוואה** בין אובייקטים. לטבלת ערבול אין פונקציית סדר בין אובייקטים. כלומר, אין כל סדר קבוע של אובייקטים, וכשסורקים טבלה, מקבלים את האובייקטים בסדר אקראי, ולא בסדר הכנסתם לקבוצה או סדר אחר כלשהו.

אחת הבעיות בטבלה כזו היא התנגשויות. אם פונקציית הערבול אינה אופטימלית, יש מספר התנגשויות רב ומספר גדול של אובייקטים מרוכז בכניסות מסוימות של הטבלה. במקרה הפחות טוב, יהיו כל האובייקטים מרוכזים ברשימה אחת, וזמן החיפוש בה יהיה לינארי.

בעיה אחרת נוצרת כשאיננו יודעים מראש את מספר האובייקטים שאנו צריכים באוסף. אם קובעים טבלה קטנה מדי, תהיינה התנגשויות רבות והחיפוש לא יהיה יעיל. אם נקבע טבלה גדולה מדי, יהיה בזבוז זיכרון ללא צורך.

אפשר למצוא שיטות לחישוב יעיל של ערך הערבול עבור סוגי משתנים שונים ומספרם. בכל מקרה, קשה מאוד למצוא פונקציית ערבול אופטימלית המחלקת את האובייקטים באופן שווה בין הכניסות השונות בטבלה.

לטבלאות קטנות ניתן להגדיר פעולה שסורקת את הטבלה ראשית, מכפילה את גודלה ומערבלת מחדש את כל האובייקטים בטבלה. פעולה כזו מקטינה את מספר ההתנגשויות ואת האורך הממוצע של כל רשימה במערך הרשימות. כתוצאה, פעולה כזו משפרת את מהירות החיפוש של אובייקטים בטבלה.

## 12.1.2.2 עצים בינאריים

מבנה נתונים אחר, בעל זמן חיפוש קצר, הוא **עץ בינארי** (Binary Tree). בעץ בינארי יש לכל צומת שני מצביעים אל הצאצאים הישירים שלו. כלומר, לכל צומת יש מצביע לתת-עץ שמאלי ולתת-עץ ימני. לעץ בינארי יש צומת מיוחד המשמש כשורש העץ, ואין לו צומת אב (הוא הראשון!) בין הצמתים של העץ מוגדר **יחס סדר** (order relation). אפשר להבחין מי גדול, או קטן יותר בין כל שני צמתים בעץ.

כל הצמתים המופיעים כבנים שמאליים של צומת מסוים קטנים מצומת האב. כל הצמתים המופיעים כבנים ימניים (תת עץ ימני) של צומת מסוים גדולים מצומת האב. הגדרה זו היא **הגדרה רקורסיבית** (recursive definition), אשר מאפשרת חיפוש יעיל בעץ. כדי למצוא צומת כלשהו יש לרדת בעץ ימינה, אם הצומת המבוקש גדול מהצומת הנוכחי. אם הצומת המבוקש קטן מהצומת הנוכחי, יש לפנות שמאלה בעץ.

הדוגמה הבאה מציגה עץ של ערכים שלמים, לשם הפשטות. בכל צומת יש מצביעים לתת עץ שמאלי וימני, והצומת (האיבר) עצמו מחזיק ערך שלם. הפונקציה tree_insert מדגימה את האלגוריתם של הכנסת אובייקט לעץ בינארי. כשרוצים להכניס צומת חדש לעץ, יש למצוא את המקום המתאים לו. יורדים בעץ עד שמגיעים לעלה (מתחילים מהשורש, כמובן). בכל רמה של העץ ייתכן שהנתון החדש גדול מהערך הנתון הנוכחי, ולכן צריך לפנות ימינה. אם הנתון החדש קטן מהנתון הנוכחי, יש לפנות שמאלה. כשמגיעים לעלה מוסיפים צומת נוסף לבן השמאלי, או הימני שלו.

```
struct node {
    node *left, *right;
    int data;
    node(int d) : data(d) { left = right = 0; }
};
tree_insert(int data)
{
    node *cur = root, *par = 0;
    while (cur) {
       par = cur;
       if (cur->data < data)
              cur = cur->right;
       else
              cur = cur->left;
    }
    if (par == 0)
       root = new node(data);
```

```
    else if (par->data < data)
      par->right = new node(data);
    else
      par->left = new node(data);
}
```

הבעיות המתעוררות בעצים הן הכנסה של צמתים המסודרים בסדר עולה, או יורד. במקרה כזה העץ מתנוון לרשימה, משום שכל הצמתים מוכנסים תמיד בתת עץ ימני, או שמאלי. במצב זה החיפוש בעץ איטי ומבנה הנתונים הופך להיות שקול לרשימה.

בעיה אחרת בעץ הבינארי, היא שיש לספק פונקציה המתארת סדר בין אובייקטים. סדר זה דומה לסדר שקיים בין מספרים, אולם הוא אינו תמיד קיים בין אובייקטים. למשל, בין מלבנים אין יחס סדר ברור. אנו לא יכולים לומר אם מלבנים שונים מקיימים את היחס קטן באופן כללי מספיק. כלומר, כאשר נתונים שני מלבנים, איננו יכולים לומר בוודאות מתי האחד קטן מהשני.

### 12.1.2.3 עץ מאוזן - Red Black Tree

**עץ מאוזן** (Balanced Tree) הוא עץ שהמרחק בו בין כל עלה אל שורש העץ זהה. בעץ כזה, המרחק בין שורש העץ לעלה הוא לכל היותר log(n), כש-n הוא מספר האובייקטים שבעץ. תכונה זו הופכת את העץ למבנה נתונים המאפשר חיפושים מהירים. כפי שכבר ראינו, חיפושים מהירים נדרשים במיוחד בפעולות על קבוצות. לפיכך, עץ הוא מבנה נתונים מתאים למימוש קבוצה.

כבר ראינו שעצים דורשים יחס סדר בין האובייקטים שמוכנסים לתוכם, כדי לקבוע את מקומם בעץ ואכן, זה אחד החסרונות במבנה נתונים זה. אך למרות החיסרון, הספרייה הסטנדרטית של C++ משתמשת בעץ בינארי מאוזן כדי לממש את הקבוצות והמילונים שכלולים בה. הספרייה משתמשת במבנה נתונים של עץ מאוזן אשר נקרא **עץ אדום-שחור** (Red Black Tree).

עץ אדום-שחור הוא עץ בינארי המקיים את התכונות הבאות:

- לכל צומת בעץ יש צבע אדום או שחור.
- לצומת אדום יש בנים שחורים בלבד.
- כל עלה בעץ הוא שחור.
- כל מסלול פשוט (ללא מעגלים) מצומת לעלה, כולל מספר זהה של צמתים שחורים.

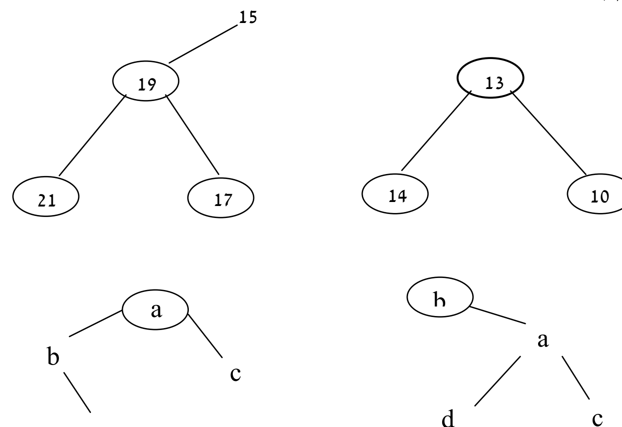
כשהעץ מקיים את התכונות האלו אפשר לראות שהמסלול הארוך ביותר משורש העץ לעלה אינו גדול מ-2*log(n), כש-n הוא מספר הצמתים בעץ ו-log הוא לפי בסיס 2. עץ אדום-שחור אינו סובל ממצב בו יש חוסר איזון מוחלט, והחיפוש בו מהיר ופרופורציונלי ל-log(n).

כדי להגיע למצב הרצוי יש **לאזן** את העץ. עושים זאת כשמוסיפים אובייקטים לעץ, או כשמבטלים אובייקטים, והדבר הרבה יותר מסובך מהכנסה של צומת לעץ, כפי שלמדנו בסעיף הקודם. נסקור פעולות אלו, אך לא נציג אלגוריתם מלא לפעולות אלה.

כשמכניסים אובייקט לעץ, מוסיפים לו למעשה צומת אדום בעל שני בנים (עלים) שחורים. מכיון שהצומת החדש הוא אדום, אנו עלולים להגיע למצב בו האב של צומת זה הוא אדום. דבר זה מתוקן בעזרת החלפת צבעים וסיבובים של העץ במידת הצורך. מכיון שמוסיפים צומת אדום, אין פגיעה במספרי הצמתים השחורים.

כשמבטלים צומת מהעץ עלולים לבטל צומת שחור, ואז מספר הצמתים השחורים במסלול מסוים עלול להיות קטן מבמסלול אחר. הפעולות האפשריות לתיקון הן החלפת צבעים וסיבוב העץ.

פעולות סיבוב של העץ גורמות לעץ להיות מאוזן יותר. האיור הבא מדגים סיבוב ימינה יחסית לצומת a של החלק השמאלי. הסיבובים מאזנים את העץ וגורמים לכך שהחלק הימני שהיה חסר בצומת יוצב לאחר הסיבוב בצד השמאלי כדי לאזן את מספר הצמתים בכל צד.




## 12.2 הקבוצה set

המחלקה של הקבוצה, **set** נמצאת ב-**STL**. לקבוצה יש איטרטור המאפשר לסרוק את האובייקטים של הקבוצה. הקבוצה עצמה מבוססת על עץ חיפוש אדום-שחור.

### 12.2.1 מבנה הקבוצה

המחלקה set ב-STL מייצגת קבוצה שבה אין חזרות, ופירוש הדבר שאובייקטים עם מפתח מסוים מופיעים בה פעם אחת בלבד. מחלקה זו משתמשת בעץ שחור-אדום, כדי לממש את הפעולות הנדרשות על קבוצה. כל צומת בעץ מחזיק העתק הנתון שהוכנס לקבוצה על ידי המשתמש.

## 12.2.2 פונקציות חשובות

יש מספר פונקציות חשובות של המחלקה set שהשימוש בהן רב ולכן נסביר את המושג **חתימה** (signature),או אבטיפוס (prototype) שלהן.

**חתימה**, או **אבטיפוס** של פונקציה, הם הפרמטרים של הפונקציה, סוגם ומספרם, וסוג הערך של הפונקציה. אלמנטים אלה מאפיינים את הפונקציה, והם הממשק שלה למשתמשים בה.

אבטיפוס של פונקציה, כלומר הפרמטרים והערך המוחזר, נראה כך למשל, בתוכנית :

```
set(const Compare& comp = Compare())
   set(const value_type* first, const value_type* last,
       const Compare& comp = Compare())

   set(const set<Key, Compare>& x) : t(x.t, false)
   set<Key, Compare>& operator=(const set<Key, Compare>&
x)
```

הבנאים מאפשרים ליצור אוביקט מסוג set. כשמגדירים אוביקט מסוג זה יש לתת לו את הפרמטרים של התבנית המייצגים את סוג האובייקט וגם פונקציה או אובייקט פונקציה, המשמשים להשוואות בין אובייקטים.

הבנאי הראשון הוא הבנאי הריק המאפשר ליצור קבוצה ריקה. הבנאי השני מאפשר ליצור קבוצה מתחום ערכים נתון. הבנאי השלישי הוא בנאי ההעתקה והאופרטור האחרון הוא אופרטור העתקה.

```
iterator begin() const { return t.begin(); }
iterator end() const { return t.end(); }
reverse_iterator rbegin() const { return t.rbegin(); }
reverse_iterator rend() const { return t.rend(); }
bool empty() const { return t.empty(); }
size_type size() const { return t.size(); }
size_type max_size() const { return t.max_size(); }
void swap(set<Key, Compare>& x) { t.swap(x.t); }
```

הפונקציות begin ו-end מחזירות איטרטורים שמתאימים לתחילת הקבוצה וסופה. הפונקציה size מחזירה את מספר האובייקטים הנוכחי בעץ. הפונקציה max_size מחזירה את גודל העץ המקסימלי.

```
pair_iterator_bool insert(const value_type& x)
```

הפונקציה insert מאפשרת הכנסת אובייקטים לעץ. הפונקציה מחזירה איטרטור שמציין אם אכן הוכנס האובייקט לעץ, או שההכנסה נכשלה. כשאובייקט כזה נמצא כבר בעץ, אין ההכנסה מתבצעת.

הפונקציה השנייה insert מאפשרת להכניס לעץ בקריאה אחת תחום נתון של אובייקטים. התחום מתחיל ב-first ומסתיים אובייקט אחד לפני last. כלומר, התחום **אינו** כולל את last.

```
void insert(const value_type* first, const value_type*
last)
void erase(iterator position)
size_type erase(const key_type& x)
void erase(iterator first, iterator last)
```

הפונקציות erase מוחקות אובייקטים מהעץ. פונקציית המחיקה הראשונה מוחקת את האובייקט המוצבע על ידי איטרטור של העץ. הפונקציה השנייה מוצאת את האובייקט המתאים לערך שהיא מקבלת ומבטלת אותו מהעץ. הפונקציה השלישית מבטלת את כל האובייקטים בעץ שנמצאים בתחום מסוים, כשהתחום נתון על ידי שני איטרטורים.

```
// set operations:

    iterator find(const key_type& x) const
    size_type count(const key_type& x) const
    iterator lower_bound(const key_type& x)
    iterator upper_bound(const key_type& x) const
    typedef  pair<iterator, iterator>
pair_iterator_iterator;
    // typedef done to get around compiler bug
pair_iterator_iterator equal_range(const key_type& x) const
```

יש מספר פונקציות המאפשרות פעולות מתמטיות על הקבוצה. הפונקציה find מוצאת אובייקט שנמצא בקבוצה. מובטח שפעולת החיפוש תהיה מהירה. סיבוכיות הזמן של פעולת החיפוש היא log(n).

הפונקציה count מונה את מספר המופעים של אובייקט נתון (במקרה זה, אפס או אחד). הפונקציה lower_bound מחזירה איטרטור המצביע לאובייקט הראשון הגדול ביותר, ש**עדיין קטן** מהאובייקט הנתון לפונקציה. הפונקציה upper_bound מחזירה איטרטור לאובייקט **הקטן** ביותר, שעדיין גדול מהאובייקט הנתון לה. הפונקציה equal_range מחזירה זוג איטרטורים המהווים את תחום האובייקט הנתון. פונקציה זו מחזירה צמד איטרטורים השקולים לאיטרטורים המוחזרים על ידי הפונקציות lower_bound ו- upper_bound. על כן, הפונקציה מחזירה תחום.

## 12.2.3 האיטרטור

איטרטור הקבוצה דומה למצביע רגיל. כשמפעילים עליו את האופרטור "*" מוחזר התוכן הנוכחי של האיטרטור, שכאן הוא האובייקט הנוכחי. איטרטור הקבוצה הוא למעשה איטרטור העץ.

איטרטורים לתחילת הקבוצה, או לסופה, מתקבלים על ידי הפונקציות begin ו-end שלה. בצורה דומה מתקבלים גם איטרטורים הפועלים בסדר הפוך, כלומר מהאובייקט הגדול לקטן. השימוש באיטרטור, במקרה זה, הוא כדי לעבור על הקבוצה. **אין** להשתמש באיטרטור עבור אלגוריתם גנרי כמו find (וזו רק דוגמה אחת). הסיבה לכך היא, שהאלגוריתם האחרון סורק את כל הקבוצה. לעומת זאת, הפונקציה find של המחלקה set פועלת ב**זמן לוגריתמי**, אשר הרבה יותר קצר מזמן ליניארי.

זאת ועוד, האיטרטור מחזיר אובייקטים קבועים כשמשתמשים באופרטור "*". הסיבה לכך היא שאסור לשנות את תכולתו, כי שינוי כזה משנה את האובייקט שנמצא בעץ. כתוצאה משינוי כזה יכול להשתנות הסדר בין האובייקט הנוכחי לאובייקטים אחרים בקבוצה, ואז מיקום האובייקט בעץ יהיה שגוי. הדבר יכול לגרום לשגיאות באלגוריתם ההכנסה, או ההוצאה של אובייקטים.

## 12.3 קבוצה עם כפילויות

**קבוצה עם כפילויות** (multiset), **קבוצה כפולה**, או **קבוצה מרובה**, דומה מאוד לקבוצה רגילה, אלא שבקבוצה כפולה מותרת החזרה על אובייקטים. כלומר, אובייקט בקבוצה כפולה יכול להופיע מספר פעמים. דבר זה נראה תמוה במבט ראשון, אך יש מצבים בהם רוצים לאפשר למספר אובייקטים, בעלי אותו מפתח, להופיע בקבוצה.

תכונת הכפילות אינה מתאימה להגדרה המתמטית של קבוצה (וגם לא יחס הסדר), ולכן חשוב שלא לעָרֵב את שני המונחים. בהגדרה המתמטית של קבוצה, אין מקום לאובייקטים חוזרים, ואין הגדרה של סדר בין האובייקטים בקבוצה. לפי ההגדרה המתמטית קבוצה היא אוסף לא סדור של אובייקטים ללא חזרות, כלומר, כל אובייקט מופיע בקבוצה פעם אחת ויחידה.

### 12.3.1 פונקציות חשובות

פונקציות **קבוצה כפולה** דומות מאוד לפונקציות של קבוצה רגילה. השינוי הגדול הוא כשמכניסים אובייקט לתוך קבוצה כפולה, ולכן לא נפרט שוב את כל הפונקציות של קבוצה כפולה.

```
iterator insert(const value_type &val)
```

במקרה זה אין כישלון בהכנסת אובייקט, כי האובייקט זהה לאובייקט שכבר נמצא בקבוצה. הפונקציה מחזירה איטרטור (מצביע) לאובייקט שהוכנס לקבוצה.

## 12.4 דוגמה - שימוש בקבוצה

בסעיף זה נראה דוגמה לשימוש בקבוצה ובקבוצה כפולה. השימוש בשתי מכולות אלו דומה מאוד (שכן הן מספקות פונקציונליות דומה), נדגים את שתי המחלקות יחד.

הדוגמה שנראה היא מימוש של **מילון** (dictionary). במילון יש מילות מפתח שלכל אחת מהן יש הסבר המבאר את פירושה. ההסבר יכול להכיל מספר רב של מילים. הדוגמה מבוססת על פונקציות גנריות (בדומה לפונקציות הגנריות של STL) שאינן תלויות בסוג הקבוצה. הדבר מאפשר לשנות את הקבוצה מקבוצה רגילה לכפולה, ולהיפך, ללא צורך בשינוי התוכנית.

יכולת הכתיבה של תוכנה בצורה גנרית כזו חשובה מאוד, כי היא גורמת לכך שהתוכנה תהיה פחות חשופה לשינויים. הקורא ייטיב לעשות אם ישנן דוגמה זו היטב.

בדוגמה אפשר להוסיף או לבטל כניסות למילון, לשנות את התיאור של כניסה מסוימת ולכתוב או לקרוא את המילון מקובץ. כמו כן, נאפשר למשתמש להציג את כל המילון. הפעולה האחרונה יכולה להיות מאוד לא ריאלית כשהמילון גדול.

הדוגמה נמצאת בדיסקט המצורף, בקבצים **dict1.cpp**, **dict1.h**.

### 12.4.1 כניסה ביומן

כל כניסה ביומן מיוצגת על ידי המחלקה DictEntry. בכל כניסה יש מחרוזת המתארת את המפתח ותיאורו (הסבר של המילה המתאימה). המחרוזת ממומשת על ידי אובייקטים מהמחלקה String שלמדנו בפרק קודם.

```
// This file includes the header for the
// dictionary using STL set. An entry in the
// dictionaty is composed of key and description.
//
#include "stl/set.h"
#include "String.H"

class DictEntry {
    String key;      // the key of this netry
    String desc;     // the key's description
public:
    DictEntry(const char *k, const char *d)
    : key(k), desc(d) {}
    const char *get_key() const
    { return key.get_cstr();}
    const char *get_desc() const
    { return desc.get_cstr(); }
    void set_desc(const char *d)
    { desc = d; }
};
```

## 12.4.2 פונקציות עזר

אנו משתמשים בקבוצה של STL (ולא בקבוצה מתמטית) ולכן יש להגדיר יחס סדר בין האובייקטים בקבוצה. נגדיר את יחס הסדר כיחס לקסיקוגרפי בין מפתחות האובייקטים במילון. כדי לממש את יחס הסדר נגדיר אובייקט פונקציה less_dict_entry המשתמש בפונקציית הספרייה strcmp, כדי להשוות בין מפתחות האובייקטים השונים.

```
// function object for the set to compare objects
struct less_dict_entry {
    int operator()(const DictEntry &de1,
            const DictEntry &de2) const
    { return (strcmp(de1.get_key(),
            de2.get_key()) < 0); }
};
```

כדי לאפשר הצגת כניסות במילון נגדיר אופרטור "<<" המקבל זרם פלט (ostream), ואובייקט המדפיס את האובייקט לזרם פלט.

```
// an operator to output an entry in the dictionary
ostream &operator<<(ostream &o, const DictEntry &de)
{
    o << de.get_key() << "\t\t" << de.get_desc() << '\n';
    return o;
}
```

בעזרת typedef אנו שולטים על סוג הקבוצה שמפעילה התוכנית. משפט ההגדרה מגדיר את המחלקה set פעם אחת כמילון עם פרמטרים של כניסה ביומן ואובייקט הפונקציה, ובפעם אחרת כקבוצה כפולה. שאר חלקי התוכנה אינם תלויים בסוג הקבוצה. גישה זו יעילה כשרוצים לשנות מבנה נתונים בעזרת הגדרה אחת, ללא כל צורך לשנות את ההתייחסות אליה בכל התוכנה.

```
// use typedef so it is easy to change between containers
// typedef set<DictEntry, less_dict_entry> Dict;
typedef multiset<DictEntry, less_dict_entry> Dict;
```

הפונקציה display פועלת בצורה המקובלת של פונקציות STL ופונקציות גנריות. היא מקבלת שני איטרטורים כפרמטרי תבנית. האיטרטור הראשון משמש להגדרת התחום שרוצים להציג, והאיטרטור השני מייצג את הפלט. הפלט יכול להיות מנותב אל הפלט הסטנדרטי (המסך), קובץ, או כל איטרטור אחר התומך באופרטור "<<".

```
// display each item on a line to a given output
template <class Iter, class Output>
void display(Iter start, Iter end, Output &out)
{
    while (start != end)
```

```
        out << *start++;
}
```

הפונקציה copy מעתיקה מחרוזת אחת לאחרת. העתקת מחרוזת המקור (src) נמשכת כל עוד לא מגיעים לתו סוף המחרוזת (0), או לתו המסמן רווח. הפונקציה מקדמת את המצביע למקור עד לתחילת המילה הבאה ומחזירה אותו. אפשר לשפר את יעילות הפונקציה, ואני משאיר זאת לקורא, כתרגיל.

```
// copy source string into destination and return a
// pointer to the source right behind the end of it.
const char *copy(char *dst, const char *src)
{
    while (*src && !isspace(*src))
        *dst++ = *src++;
    *dst = 0;
    while (*src && isspace(*src))
        ++src;
    return src;
}
```

הפונקציה getline היא פונקציית תבנית המאפשרת קבלת שורת קלט ממחלקות התומכות בפעולה get, אשר מחזירה את תו הקלט העוקב. השורה נקראת עד לתו הזיהוי של שורה חדשה, המוכנס לחוצץ שהארגומנט line מתייחס אליו. תו הזיהוי של שורה חדשה מוחלף בערך המספרי אפס, המציין את סוף מחרוזת.

```
// this function reads the input until a new line
// char or until the input buffer is exauseted.
// The first new line is discareded.
template <class Input>
void getline(Input &in, char *line, int size)
{
    const char nl = '\n';
    char *end = line + size - 1;
    in.get(*line);
    if (*line != nl)
        ++line;
    while (line < end) {
        in.get(*line);
        if (*line == nl)
            break;
        else
            ++line;
    }
    *line = 0;
}
```

פונקציית התבנית input_entry, בצירוף פונקציות נוספות, מאפשרת קבלת קלט שבו מילת מפתח אחת, ומספר מילות הסבר למילת המפתח שנגמרות בשורה חדשה. הפונקציה מחזירה אובייקט שמייצג כניסה אחת ביומן. פונקציית תבנית זו מחזירה אובייקט אל מחסנית התוכנית, וכך האובייקט המוחזר מועתק לתוך האובייקט המבוקש. העתקה אינה פעולה זולה וניתן למנוע אותה על ידי העברת אובייקט לפונקציה זו, שתאתחל אותו.

```
// inputs one entry of the dictionary and returns it
// on the stack.
template <class Input>
DictEntry input_entry(Input &in)
{
     const int bufsize = 256, linesize = 1024;
     char key[bufsize], desc[bufsize],
          line[linesize];

     getline(in, line, linesize);
     const char *src = copy(key, line);
     strcpy(desc, src);
     return DictEntry(key, desc);
}
```

פעולה אחרת, הנדרשת במילון היא עדכון של תיאור מילת מפתח. תיאור מילת המפתח ניתן לשינוי, ללא ביטול המילה והכנסה של מילה אחרת למילון, כי תיאור זה אינו משתתף ביחס הסדר. לעומת זאת, **אסורה** פעולת שינוי של מילת מפתח.

הפונקציה מחפשת את הכניסה המתאימה בעזרת פונקציית החיפוש של הקבוצה ומחזירה איטרטור. אם האיטרטור שונה מהאיטרטור המוחזר על ידי הפונקציה end של הקבוצה, הרי שהכניסה המתאימה נמצאה, ואפשר לעדכן אותה. אחרת, לא קיימת כניסה כזו.

```
// update one entry. It updates the description of
// the entry. You would get a compiler error since
// the iterator returns a const object for the
// * operator. since we do not update the key it is ok.
void update_dict(Dict &d)
{
    DictEntry de = input_entry(cin);
    Dict::iterator i = d.find(de);
    if (i != d.end())
       (*i).set_desc(de.get_desc());
    else
      cout << "entry not found" << endl;
}
```

הפונקציה המבטלת כניסה מהמילון משתמשת במילת המפתח כדי לעשות זאת. באמצעות מילת המפתח שמכניס המשתמש, בונה הפונקציה כניסה ביומן (עם תיאור ריק), ומשתמשת בפונקציה erase כדי לבטל את הכניסה המתאימה.

```
// removes an entry from the dictionary
void remove_ent(Dict &d)
{
    char key[64];
    cout << "\nEnter key to remove:";
    cin >> key;
    cout << endl;
    DictEntry de(key, "");
    d.erase(de);
}
```

הפונקציות read_dict ו-write_dict משתמשות בפונקציה get_file_name כדי לקבל את שם הקובץ שממנו יש לקרוא, או שלתוכו יש לכתוב. הפונקציה write_dict פשוטה, משום שהיא משתמשת בפונקציה display כדי לכתוב את המילון לקובץ נתון.

הפונקציה display משמשת לכתיבה לקובץ ולהצגת אובייקט על הפלט הסטנדרטי. הפונקציה מיישמת את רעיון התכנות הגנרי, **עבודה על תחום הנתון על ידי איטרטורים**, ולכן היא טובה לכל תחום. הפלט אף הוא פרמטר של התבנית שלה, ולכן ניתן לשנות אם את סוגו!

```
// get a file name from the user
const char *get_file_name()
{
    static char buf[128];
    cout << "\nenter file name:";
    cin >> buf;
    cout << endl;
    return buf;
}

// write the dictionary into a file.
void write_dict(Dict &d)
{
    const char *file_name = get_file_name();
    ofstream out(file_name);
    display(d.begin(), d.end(), out);
}
```

הפונקציה read_dict מסובכת יותר מהפונקציה write_dict. הפונקציה מגדירה אובייקט מסוג ifstream. אם האובייקט תקין (כלומר, קיים קובץ מתאים), הפונקציה מנקה את המילון בלולאה שבה היא מקבלת כניסה אחר כניסה מהקובץ הנתון בעזרת הפונקציה input_entry. הפונקציה input_entry משמשת לקבלת כניסה ביומן,

מהמשתמש או מקלט. השימוש בפונקציית תבנית מאפשר שליטה על סוג הקלט, ללא צורך בשינוי פונקציית התבנית, או כתיבת פונקציות נוספות.

```
// read a dictionary from a file. If the given
// file exist erases the dictionary and then
// reads input into the file using a while
// loop
void read_dict(Dict &d)
{
    const char *file_name = get_file_name();
    ifstream in(file_name);
    if (in) {
         d.erase(d.begin(), d.end());
         while (in) {
              DictEntry de = input_entry(in);
              if (in)
                   d.insert(de);
         }
    }
}
```

הפונקציה display_menu מציגה למשתמש תפריט פעולות אפשריות, ומקבלת ממנו את הפעולה המבוקשת. מספר הפעולה המבוקשת מוחזר לפונקציה שקוראת לפונקציה זו.

```
enum commands { exit_cmd, insert_cmd, display_cmd,
                update_cmd, remove_cmd, write_cmd,
                read_cmd };

// displays a menu to the user and returns the command
int display_menu()
{
   int res = exit_cmd;
   cout << exit_cmd << ") exit.\n";
   cout << insert_cmd << ") insert a new entry.\n";
   cout << display_cmd << ") display dictionary contents.\n";
   cout << update_cmd << ") update a key.\n";
   cout << remove_cmd << ") remove a key.\n";
   cout << write_cmd << ") write into a file.\n";
   cout << read_cmd << ") read from a file.\n";
   cout << "> ";
   cin >> res;
   return res;
}
```

הפונקציה הראשית מגדירה אובייקט מסוג מילון ופועלת על אובייקט זה בלולאה. היא מקבלת את הפעולה המבוקשת מהפונקציה display_menu ומבצעת את הפעולה הדרושה בעזרת קריאה לפונקציות העזר, ובאופן ישיר על המילון. הפונקציה יוצאת מהלולאה כשמתקבלת פקודת היציאה מהמשתמש.

```
// the main function just displays the menu and
// executes the commands from the user
int main()
{
    Dict dict;
    DictEntry de("", "");
    int cmd;
    while ((cmd = display_menu()) != exit_cmd) {
         switch (cmd) {
         case insert_cmd:
              cout << "\nEnter key and description:";
              de = input_entry(cin);
              dict.insert(de);
                   break;
         case display_cmd:
              display(dict.begin(),
                   dict.end(), cout);
              cout << endl;
                   break;
         case update_cmd:
              update_dict(dict);
                   break;
         case remove_cmd:
              remove_ent(dict);
                   break;
         case write_cmd:
              write_dict(dict);
                   break;
         case read_cmd:
              read_dict(dict);
                   break;
         case exit_cmd:
              default:
                   break;
         }
    }
    return 0;
}
```

## 12.5 מפות (multimap ,map)

המחלקות **map** ו-**multimap** נקראות **מפות, מילונים**, או בעגה המקצועית: **מערכים אסוציאטיביים** (associative arrays). העיקרון העומד מאחורי מחלקות אלו הוא **קישור בין מילות מפתח לערכים**. כל כניסה במכולה מסוג זה היא של זוג (Key, Value). מבני נתונים אלה אידיאליים עבור בעיות כמו המילון שראינו בסעיפים הקודמים.

### 12.5.1 מפה (map)

המחלקה **map** מאפשרת לחבר בין מפתח לערך, וגם לגשת במהירות לערכים לפי מפתחות. במפה יכול להיות מפתח אחד בלבד. מפתח בעל ערך מסוים יכול להופיע פעם אחת בלבד והחזרות אסורות.

מפה היא מחלקת תבנית, וכשמגדירים מפה יש לספק את פרמטרי התבנית הבאים: סוג המפתח, סוג הערכים ופונקציית סדר בין המפתחות. פונקציית הסדר יכולה להיות אובייקט פונקציה שמספקת פונקציה זו.

המפה מעתיקה לצמתים שלה את הערכים הנמסרים לה, ולכן היא מכילה העתקים של המפתחות והערכים שנמסרו לה. המפה גם אחראית להקצאה ושחרור של אובייקטים הנמצאים בה.

ניתן להגדיר מפה של מצביעים. במקרה כזה, מוטלת עלינו האחריות להקצאת זיכרון למצביעים ולשחרור הזיכרון.

#### 12.5.1.1 פונקציות חשובות

הפונקציות החשובות ביותר של מפה הן **בנאי המחלקה**, **פונקציות הכנסה וביטול** של אובייקטים **ופונקציות חיפוש** של מפתח.

הבנאי הראשון הוא בנאי ברירת המחדל, שמקבל אובייקט המציין את יחס הסדר בין המפתחות. לבנאי יש ברירת מחדל, שהוא אובייקט יחס סדר שנוצר ללא פרמטרים.

הבנאי השני מאפשר לאתחל את המפה מתחום של ערכים נתונים, שבמקרה זה הם זוגות המוגדרים בעזרת המבנה pair שלו שני שדות, ראשון ושני, המהווים את הזוג. השדה הראשון הוא המפתח, והשני הוא הערך.

```
map(const Compare& comp = Compare())
map(const value_type* first, const value_type* last,
      const Compare& comp = Compare())
map(const map<Key, T, Compare>& x)

template <class T1, class T1>
```

```
struct pair {
      T1 first;
      T2 second;
pair(T1 &v1, T2 &v2) : first(v1), second(v2) {}
};
```

במקור, קיבל הבנאי של pair ייחוס קבועים לאובייקטים. כתוצאה מכך, אם יגדירו את הזוג בדרך הבאה -

```
pair<const String, String> value_type;
```

לא יוכל המהדר להסתדר עם ההגדרה הכפולה של const. לכן, צריך להמיר את הגדרת הבנאי כפי שמתואר כאן.

הבנאי האחרון הוא בנאי ההעתקה שמאפשר להגדיר מפה כהעתק של מפה אחרת. בנוסף לבנאי ההעתקה, מסופק גם אופרטור העתקה המאפשר להעתיק מפה אל מפה. שניהם מוגדרים תמיד לכל מחלקה. אם המתכנת אינו מספק אותם מייצר המהדר בנאי ואופרטור העתקה. כאשר המהדר מייצר אותם, הוא מבצע העתקה רדודה, שבמקרים רבים (כמו במקרה זה) אינה מספקת. כאשר יש מצביעים, יהיו האופרטורים שמספק המהדר אינם מספקים בדרך כלל. במקרים שמחלקה מכילה מצביעים, ו/או שהסמנטיקה של העתקה רדודה אינה מספקת, צריך להגדיר **אופרטור השמה ובנאי העתקה**.

```
map<Key,T,Compare>& operator=(const map<Key,T,Compare>& x)
```

למחלקה יש מספר פונקציות המספקות איטרטורים. כאן, האיטרטורים שמקבלים אינם מאפשרים לשנות את המפתח. הסיבה לכך היא שבמקרה של שינוי במפתח צריך להשתנות מיקום הצמד (pair) בעץ. כלומר, יש להוציא את הצמד מהעץ ולהכניסו שוב.

אנו מקבלים שני איטרטורים. הראשון הוא כזה שמאפשר לעבור על המפה בכיוון חיובי (ממפתח קטן לגדול יותר), והשני מאפשר לעבור על המפה בכיוון השלילי (ממפתח גבוה לנמוך יותר).

הפונקציה begin מחזירה איטרטור לתחילת המפה, והפונקציה end מחזירה איטרטור לאובייקט אחד אחרי האובייקט האחרון במפה. מכיון שהאובייקטים אינם מסודרים בזיכרון רציף, אין אפשרות לבדוק אם אופרטור אחד קטן מחברו, כמו שאנו נוהגים במצביעים רגילים.

```
iterator begin()
const_iterator begin() const
iterator end()
const_iterator end() const
reverse_iterator rbegin()
const_reverse_iterator rbegin() const
reverse_iterator rend()
const_reverse_iterator rend() const
```

הפונקציה empty מחזירה ערך בוליאני אמת אם המכולה (המפה) ריקה, ואחרת (המכולה אינה ריקה) - ערך אפס, השקול ל-false.

```
bool empty() const
```

הפונקציה, המכניסה צמד אובייקטים למפה, מחזירה צמד: איטרטור וערך בוליאני. אם ההכנסה הצליחה, כלומר, לא קיים צמד עם מפתח זהה במפה, הערך הבוליאני הוא אמת (השדה השני בצמד), ואחרת הערך הוא false. במקרה של הצלחה - הערך הראשון הוא איטרטור המקום בקבוצה שאליו הוכנס הצמד.

פונקציית ההכנסה השנייה מאפשרת להכניס תחום נתון לתוך הקבוצה. התחום נתון על ידי שני מצביעים לצמדים (או זוגות). כשהמצביע השני מצביע למקום אחד אחרי האובייקט האחרון שרוצים להכניס למפה.

```
typedef pair<iterator, bool> pair_iterator_bool;
      // typedef done to get around compiler bug
pair_iterator_bool insert(const value_type& x)
void insert(const value_type* first,const value_type* last)
```

פונקציות הביטול (erase) מאפשרות לבטל אובייקטים מהמפה. פונקציית המחיקה הראשונה מוחקת את הצמד המוצבע על ידי האיטרטור הנתון לה. פונקציית המחיקה השנייה מוחקת צמד שהמפתח שלו זהה למפתח הנתון. פונקציית המחיקה השלישית מוחקת את כל האובייקטים בעלי מפתחות זהים לאלה הנתונים בתחום (first, last).

```
void erase(iterator position)
size_type erase(const key_type& x) { return t.erase(x); }
void erase(iterator first, iterator last)
```

הפונקציות הבאות מנצלות את מהירות החיפוש הגבוהה של עצים מאוזנים ומספקות פעולות שכיחות על מפה.

הפונקציה find מוצאת זוג המתאים למפתח נתון ומחזירה איטרטור המצביע אליו. אם האיטרטור המוחזר זהה לאיטרטור המוחזר על ידי הפונקציה end, פירוש הדבר שנמצא זוג שמתאים למפתח הנתון.

הפונצקיה count מאפשרת למנות את מספר המופעים (אפס או אחד) של מפתח נתון במפה. הפונקציה lower_bound מחזירה איטרטור לאובייקט הגדול ביותר שקטן מהמפתח הנתון לה. הפונקציה upper_bound מחזירה איטרטור לאובייקט הקטן ביותר שגדול מהמפתח הנתון לה. הפונקציה equal_range מחזירה צמד איטרטורים שהראשון הוא כמו האיטרטור המוחזר מ-lower_bound והשני הוא כמו זה המוחזר מ-upper_bound.

```
// map operations:

  iterator find(const key_type& x)
  const_iterator find(const key_type& x) const
  size_type count(const key_type& x) const
  iterator lower_bound(const key_type& x)
```

```
  const_iterator lower_bound(const key_type& x) const
  iterator upper_bound(const key_type& x)
  const_iterator upper_bound(const key_type& x) const
  typedef pair<iterator, iterator> pair_iterator_iterator;
  // typedef done to get around compiler bug
  pair_iterator_iterator equal_range(const key_type& x)
};
```

### 12.5.1.2 האיטרטור

האיטרטור מתנהג כמצביע לזוגות. זוג כזה מכיל, בשדה הראשון את המפתח, ובשני את הערך. כשמשתמשים באיטרטור אסור לשנות את ערך המפתח, כי שינוי זה גורם להתנהגות בלתי מוגדרת באלגוריתמי הכנסה והוצאה של זוגות מהעץ.

איטרטור המפה הוא איטרטור דו-כיווני, אך אינו איטרטור גישה אקראית. לכן, איטרטור זה אינו תומך בפעולות "<" או ">", והוא אין אפשרות לקדם אותו בקפיצות השונות מאחד. הקידום של האיטרטור אפשרי רק לצמד הבא, או לצמד הקודם לנוכחי.

## 12.5.2 מפה כפולה (multimap)

**מפה כפולה** (multimap) היא מחלקה המאפשרת לקשר בין מפתחות לערכים. גם מחלקה זו היא מחלקת תבנית, כשפרמטרי התבנית מגדירים את סוג המפתחות, סוג הערכים ופונקציה (או אובייקט פונקציה) המגדירה את הסדר בין המפתחות.

במפה כפולה אין הגבלה על מספר המפתחות הזהים, כלומר, מותר להכניס למפה כפולה מספר רב של צמדים עם מפתחות זהים. כמובן שאפשר להכניס מספר צמדים עם אותו מפתח, אך עם ערך שונה.

### 12.5.2.1 פונקציות חשובות

הפונקציות של מפה כפולה זהות לפונקציות של מפה רגילה, ולא נחזור עליהן כאן. השוני הוא שבמפה כפולה אפשר להכניס מספר לא מוגבל של צמדים הכוללים מפתח זהה. פעולת ההכנסה אינה נכשלת גם אם קיים במפה צמד בעל מפתח הזהה למפתח הצמד המועמד להכנסה למפה, ומחזירה איטרטור לצמד שהוכנס לקבוצה.

### 12.5.2.2 האיטרטור

איטרטור המפה הכפולה הוא דו-כיווני, המאפשר לסרוק את המפה בשני כיוונים: ממפתח קטן לגדול, או להיפך. האיטרטור אינו איטרטור לגישה אקראית ולכן אין אפשרות לקדם אותו, או להפחית ממנו מספר השונה מאחד. האיטרטור מאפשר להתקדם או לסגת בצעד אחד בלבד.

אסור לשנות את ערך המפתח של הצמד עליו מצביע האיטרטור. שינוי ערך המפתח משפיע על מיקום הצמד בעץ. האיטרטור אינו מונע זאת, אך במקרה של שינוי ערך המפתח תהיה התנהגות בלתי מוגדרת לפונקציות ההכנסה, או הביטול של אובייקטים.

האיטרטור מתנהג כמו מצביע, וכשמפעילים עליו את האופרטור "*" הוא מחזיר ייחוס לצמד הנוכחי.

## 12.5.3 דוגמה - שימוש במפה

כעת נציג דוגמה לשימוש במפה ובמפה כפולה. הדוגמה נמצאת בדיסקט המצורף בקובץ **dict2.cpp**. הדוגמה אינה תלויה בסוג המפה בה נשתמש. כשנרצה לשנות את סוג המפה עלינו לתקן את הפקודה typedef כנדרש. הדוגמה עוסקת במילון (כמו הדוגמה הקודמת) שבו יש תיאור של מילות מפתח. במקרה זה, ההתאמה לדוגמה היא ישירה, מכיון שהמפה מכילה צמדים. איננו צריכים להגדיר כניסה במילון, כי היתרון של מפה על קבוצה הוא בכך שאין צורך להגדיר מחלקת עזר המייצגת כניסה במילון.

### 12.5.3.1 פונקציות עזר

המפה מבוססת על עץ בינארי, ולכן עלינו לספק פונקציה, או אובייקט פונקציה, לביצוע ההשוואה הדרושה בין אובייקטי המפה. כלומר, השוואה בין המפתחות הנמצאים במפה. נספק אובייקט פונקציה שיבצע את ההשוואה הנדרשת. זהו אובייקט עם אופרטור הקריאה לפונקציה "()".

```
// chapter 12: dictionary program using map and multimap
// function object for the set to compare objects
struct less_dict_entry {
    int operator()(const String &s1,
       const String &s2) const
    { return (strcmp(s1.get_cstr(), s2.get_cstr())
         < 0); }
};
```

כדי לאפשר שינוי של התוכנית מסוג מכולה אחד לאחר, נגדיר את סוג המילון בעזרת typedef. כמו כן, נשתמש בהגדרת המילון של הסוג value_type כדי להגדיר בו כניסות. בדרך זו, כשמשתנה סוג המילון, משתנה גם הגדרת הכניסה במילון. כדי לשנות את שני הפרטים האחרונים, עלינו לשנות את הגדרת סוג המילון בלבד. דבר זה נעשה על ידי הפיכת אחת משורות typedef להערה, וביטול ההערה בשורה האחרת.

כדי לאפשר פלט של אובייקטים מסוג String, מוגדר האופרטור הגלובלי "<<" המקבל ייחוסים לאובייקט מסוג ostream ולאובייקט מסוג String. האופרטור מחזיר ייחוס לאובייקט הפלט, כדי שאפשר יהיה לשרשר בין פעולות פלט.

```
// use typedef so it is easy to change between containers
typedef map<String, String, less_dict_entry> Dict;
//typedef multimap<String, String, less_dict_entry> Dict;

typedef Dict::value_type dict_entry;

// an operator to output an entry in the dictionary
ostream &operator<<(ostream &o, const String &s)
{
      o << s.get_cstr();
      return o;
}
```

הפונקציה display מאפשרת להדפיס תחום המוגדר בעזרת איטרטורים. הפונקציה מניחה שתכולת האיטרטורים היא צמדים, ולכן על איטרטורים שאינם מכילים צמדים נקבל שגיאה מהמהדר במהלך הידור תוכנית המקור. כעיקרון, עדיף לקבל שגיאת הידור על פני שגיאת זמן ריצה, מכיון שיותר קשה לגלות ולטפל בשגיאת זמן ריצה.

```
// display each item on a line to a given output
template <class Iter, class Output>
void display(Iter start, Iter end, Output &out)
{
    while (start != end) {
         out << (*start).first << "\t\t\t"
              << (*start).second << '\n';
         ++start;
    }
}
```

### 12.5.3.2 פונקציות קלט

מספר פונקציות משמשות לקלט. הפונקציות copy, getline ו-input_entry מאפשרות קבלת כניסה ביומן מהמשתמש בתוכנית. הפונקציות copy ו-getline זהות לדוגמה הקודמת, ולא נרחיב עליהן את הדיבור. הפונקציה input_entry מקבלת אובייקט פלט, שסוגו הוא פרמטר של התבנית, קוראת שורה שלמה מהקלט, ומכניסה את השורה לשני אובייקטי המחרוזת שהיא מקבלת.

```
// copy source string into destination and return a
// pointer to the source right behind the end of it.
const char *copy(char *dst, const char *src)
{
    while (*src && !isspace(*src))
       *dst++ = *src++;
```

```
    *dst = 0;
    while (*src && isspace(*src))
        ++src;
    return src;
}

// this function reads the input until a new line char
// or until the input buffer is exauseted.
// The first new line is discareded.
template <class Input>
void getline(Input &in, char *line, int size)
{
    const char nl = '\n';
    char *end = line + size - 1;
    in.get(*line);
    if (*line != nl)
        ++line;
    while (line < end) {
        in.get(*line);
        if (*line == nl)
              break;
        else
              ++line;
    }
    *line = 0;
}

// inputs one entry of the dictionary
// and returns it on the stack.
template <class Input>
void input_entry(Input &in, String &k, String &d)
{
        const int bufsize = 256, linesize = 1024;
        char key[bufsize], desc[bufsize];
    char line[linesize];

        getline(in, line, linesize);
        const char *src = copy(key, line);
        strcpy(desc, src);
    k = key;
    d = desc;
}
```

## 12.5.3.3 פונקציות עדכון למילון

הפונקציות update_dict ו-remove_ent מעדכנות את המילון. הראשונה, מעדכנת את ההסבר הצמוד למילת מפתח נתונה, והשנייה מבטלת כניסה המתאימה למילת מפתח נתונה.

הפונקציה update_dict משתמשת בפונקציית הקלט input_entry כדי לקבל מפתח, ותיאור שלו. אם המפתח כבר נמצא ביומן, מחזירה הפונקציה find איטרטור המצביע על הכניסה המתאימה, ואחרת היא מחזירה איטרטור הפונה לסוף המילון. סוף המילון הוא איטרטור אחד מעבר לאובייקט האחרון שנמצא במילון. אם נמצאת כניסה מתאימה במילון, מעודכנת הכניסה בהתאם לתיאור הנתון.

```
// update one entry. It updates the description of the
// entry. You would get a compiler error since the
// iterator returns a const object for the * operator.
// since we do not update the key it is ok.
void update_dict(Dict &d)
{
    String key, desc;
    cout << "Enter a key and description:";
    input_entry(cin, key, desc);
    cout << endl;
    Dict::iterator i = d.find(key);
    if (i != d.end())
       (*i).second = desc.get_cstr();
    else
      cout << "entry not found" << endl;
}
```

הפונקציה remove_ent מקבלת מהמשתמש מחרוזת שמתאימה לכניסה עם מפתח אשר זהה למחרוזת הנתונה, ושיש להרחיק (למחוק) מהיומן.

```
// removes an entry from the dictionary
void remove_ent(Dict &d)
{
    char key[64];
    cout << "\nEnter key to remove:";
    cin >> key;
    cout << endl;
    String de(key, "");
    d.erase(de);
}
```

## 12.5.3.4 קלט ופלט מקבצים

קלט ופלט מקבצים מתבצע בעזרת הפונקציות get_file_name, read_dict ו-write_dict. שתי הפונקציות הקוראות וכותבות לקובץ משתמשות בפונקציה get_file_name, כדי לקבל מהמשתמש את שם הקובץ המבוקש. הפונקציה write_dict משתמשת בפונקציה display, כדי לכתוב את המילון לקובץ.

הפונקציה read_dict, מסובכת יותר. אם הפונקציה מצליחה לפתוח את הקובץ המבוקש היא מבטלת את כל האובייקטים במילון. לאחר מכן, באמצעות הפונקציה input_entry נקראות הכניסות, זו אחר זו, ומוכנסות למילון. לאחר קריאת כניסה נבדק אובייקט הקלט, אם הוא עדיין תקין. הדבר נעשה כדי למנוע הוספה של אובייקט לא מוגדר, כשמגיעים לסוף הקובץ.

```
// get a file name from the user
const char *get_file_name()
{
    static char buf[128];
    cout << "\nenter file name:";
    cin >> buf;
    cout << endl;
    return buf;
}

// write the dictionary into a file.
void write_dict(Dict &d)
{
    const char *file_name = get_file_name();
    ofstream out(file_name);
    display(d.begin(), d.end(), out);
}

// read a dictionary from a file. If the given file
// exist erases the dictionary and then reads input
// into the file using a while loop
void read_dict(Dict &d)
{
    const char *file_name = get_file_name();
    ifstream in(file_name);
    if (in) {
       d.erase(d.begin(), d.end());
       while (in) {
             String key, desc;
             input_entry(in, key, desc);
                         dict_entry val(key, desc);
```

```
                        if (in)
                              d.insert(val);
            }
      }
}
```

### 12.5.3.5 פונקציות הניהול

פונקציות הניהול מציגות למשתמש תפריט ומבצעות את הפקודה המבוקשת על ידו. הפקודות שהמשתמש יכול להכניס (לבחור) מפורטות בפקודה enum. הפונקציה display_menu מציגה את תפריט האפשרויות ומקבלת את האפשרות המבוקשת על ידי המשתמש. אפשרות זו מוחזרת כערך שלם לפונקציה שקראה לה.

הפונקציה הראשית main קוראת לפונקציה display_menu כדי לקבל את הפקודה המבוקשת על ידי המשתמש. הערך השלם המוחזר הוא אחד הערכים המפורטים ב-enum, ולפיו מתבצעת הפעולה המתאימה. כדי לבצע את הפעולה המבוקשת משתמשת הפונקציה הראשית במשפט switch.

```
enum commands { exit_cmd, insert_cmd, display_cmd,
             update_cmd, remove_cmd, write_cmd, read_cmd
};

// displays a menu to the user and returns the command
int display_menu()
{
   int res = exit_cmd;
   cout << exit_cmd << ") exit.\n";
   cout << insert_cmd << ") insert a new entry.\n";
   cout << display_cmd << ") display dictionary contents.\n";
   cout << update_cmd << ") update a key.\n";
   cout << remove_cmd << ") remove a key.\n";
   cout << write_cmd << ") write into a file.\n";
   cout << read_cmd << ") read from a file.\n";
   cout << "> ";
   cin >> res;
   return res;
}

// the main function just displays the menu and
// executes the  commands from the user
int main()
{
      Dict dict;
      String key, desc;
```

```
	int cmd;
	dict_entry val(key, desc);
	while ((cmd = display_menu()) != exit_cmd) {
		switch (cmd) {
		case insert_cmd:
			cout << "\nEnter key and"
			cout << " description:";
			input_entry(cin, key, desc);
			val.first = key;
			val.second = desc;
			dict.insert(val);
			break;
		case display_cmd:
			display(dict.begin(), dict.end(), cout);
			cout << endl;
			break;
		case update_cmd:
			update_dict(dict);
			break;
		case remove_cmd:
			remove_ent(dict);
			break;
		case write_cmd:
			write_dict(dict);
			break;
		case read_cmd:
			read_dict(dict);
			break;
		case exit_cmd:
		default:
			break;
		}
	}
	return 0;
}
```

## 12.6 סיכום

בפרק זה למדנו על המכולות של הספרייה הסטנדרטית STL שערוכות כעץ חיפוש בינארי. המכולות יעילות במיוחד לפעולות חיפוש שסיבוכיות הזמן שלהן היא log(n). המחיר ש"משלמים" ביצירת עץ כזה הוא שלושה מצביעים לכל כניסה, ולכן אם הכניסות קטנות, התשלום גבוה יחסית. דבר נוסף שחייבים להגדיר הוא יחס סדר של האובייקטים שבמכולה. יחס הסדר ניתן להגדרה בעזרת פונקציה, או אובייקט פונקציה. אובייקטי הפונקציה שהגדרנו, זהים לאובייקט הפונקציה less שמסופק על ידי הספרייה (STL).

המכולות המייצגים קבוצה הן: **set** ו-**multiset**. הן מקבלות אובייקטים שעליהן להכניס למכולה. המכולה הראשונה אינה מאפשרת אובייקטים כפולים (אוסף ללא חזרות), והשנייה מאפשרת חזרות של אובייקטים.

המכולות המייצגות מפות הן: **map** ו-**multimap**. הראשונה מייצגת מפה ללא חזרות והשנייה מייצגת מפה עם חזרות. מפות הן מכולות המאפשרות להחזיק צמדי אובייקטים. הערך הראשון בצמד הוא המפתח שקובע את מיקום הצמד במכולה, והשני הוא הערך הקשור למפתח. במקרה של מילון מילים, זה תיאור המפתח.

בדוגמאות שראינו היו שתי גרסאות של מילונים. במקרה זה, האובייקטים המייצגים מפות קלים יותר לשימוש. גם למדנו כיצד לכתוב פונקציות גנריות המאפשרות לקרוא, או לכתוב אובייקטים לכל פלט.

הגישה הגנרית היא גישה סטטית. כלומר, הכול נבדק ונוצר על ידי המהדר בזמן ההידור, בניגוד לפונקציות וירטואליות, שנקראות על פי סוג האובייקט בזמן הריצה. קוד המשתמש בתבניות גדול יותר וצורך יותר זיכרון. לעומת זאת, קוד כזה מהיר יותר מקוד המשתמש בפונקציות וירטואליות. C++ מאפשרת לנו לבחור באחת משיטות אלו, או בשתיהן.

תכנות גנרי מאפשר יעילות גבוהה במהירות ויעילות סבירה בזמן. כשעובדים בשיטה זו, יש לזכור שהמהדר יוצר קוד לכל מחלקת תבנית ופונקציית תבנית בזמן השימוש בהן. לכן, אם אפשר להגדיר חלק מהעבודה במחלקה רגילה או בפונקציות רגילות - הרווח גדול.

## 12.7 שאלות

1. ממש טבלת ערבול יעילה וממש את המחלקה set בעזרת טבלת הערבול.
2. שנה את הדוגמאות בפרק זה כך שלא יהיה צורך להגדיר בעזרת typedef את סוג המילון.
3. שפר את יעילות הפונקציה copy (רמז: התייחס לפעולות קידום המצביעים).
4. ממש את הפונקציה copy כפונקציית תבנית.

## פרק 13

# שילוב סוגי תכנות

בפרק זה נלמד שני נושאים עיקריים: **שילוב בין תכנות גנרי לבין תכנות מוכוון אובייקטים**, ו**מתאמים** (adaptors) בספרייה הסטנדרטית. בדוגמאות שלפנו מהספרייה הסטנדרטית (**STL**) עד כה, ראינו תכנות גנרי המבוסס על תבניות ++C. השילוב בין תכנות גנרי לבין תכנות מוכוון אובייקטים הוא כלי רב עוצמה.

**מתאמים** (**adaptors**) הם מחלקות המשתמשות במחלקה אחרת כדי להתאים אותה לפונקציונליות נדרשת. בספריית התבניות הסטנדרטית יש מתאמים כאלה. המונח **מתאם** מקובל כיום בתכנות מוכוון אובייקטים, ולכן חשוב להכירו.

בפרק זה נראה דוגמה המשלבת מתאמים, תכנות גנרי ותכנות מוכוון אובייקטים.

## 13.1 מתאמים (Adaptors)

ה**מתאם** (adaptor) הוא אובייקט המתאים לממשק של אובייקט אחר כדי לממש פונקציונליות נדרשת. בדרך כלל, הוא אינו מכיל אלגוריתמים רבים, אלא משתמש באלגוריתמים של אובייקטים אחרים כדי לממש את הפונקציונליות שלו. מתאמים דרושים לנו במספר מקרים: מקרה אחד הוא תיאום בין ספריות שונות ומקרה אחר הוא המרת ממשק של אובייקט כדי לקבל פונקציונליות נדרשת. בסעיף זה נלמד על שני מקרים אלה.

### 13.1.1 תיאום בין ספריות שונות

אחד המקרים בהם נפעיל מתאמים הוא כאשר צריך לתאם בין ספריות שונות. כאשר יש שתי ספריות שונות שמצפות לקבל ממשק שונה, אנו יכולים לכתוב מתאם שממיר קריאות פונקציונליות של ספרייה אחת לאחרת.

נניח למשל, שיש פונקציה כלשהי בספרייה קלט-פלט שקוראת קובץ ומכניסה את האובייקטים שלו לרשימה נתונה. נניח שהספרייה מגדירה מחלקה בסיסית מופשטת של רשימה, באופן הבא:

```
class IOList {
public:
    IOList() {}
    virtual ~IOList() {}
    virtual void add(IOObject *);
};

void io_read_file(const char *fname, IOList &);
```

עוד נניח, שנתונה לנו ספרייה של מכולות (כמו STL) המספקת רשימה. ידוע שהרשימה בספרייה הסטנדרטית אינה מספקת פונקציה כזו. נוסף על כך, הפונקציה שקוראת קובץ מצפה לקבל ייחוס לאובייקט מסוג IOList, או אובייקט שנגזר ממחלקה זו. במצב זה אין אנו רוצים לכתוב רשימה חדשה, כי הדבר יצרוך זמן כתיבה וניפוי. ניתן להגדיר את המחלקה הבאה:

```
class List : public IOList {
    list<IOObject*> list;
public:
    List() {}
    void add(IOObject *obj)
    { list.push_back(obj); }
    ...
};
```

המחלקה List יורשת מהמחלקה IOList, ולכן נשתמש במתאם (List) כארגומנט לפונקציה io_read_file. איננו כותבים קוד נוסף, אלא משתמשים בקוד הקיים. אם נרצה להוסיף פונקציונליות של הרשימה list, עלינו להוסיף זאת בממשק המחלקה שלנו. למשל:

```
class List : public IOList {
    list<IOObject*> list;
public:
    List() {}
    void add(IOObject *obj)
    { list.push_back(obj); }
    void push_back(IOObject *obj)
    { list.push_back(obj); }
    void push_front(IOObject *obj)
    { list.push_front(obj); }
    ...
};
```

הוספת פונקציות לממשק הרשימה מחייב קריאה לפונקציה המתאימה ברשימה של STL. כדי למנוע קריאות לפונקציות אלו, ניתן לשנות את המחלקה המתאמת כך:

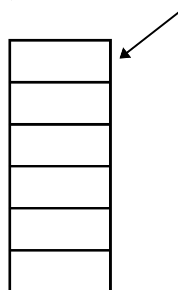
```
class List : public IOList, public list<IOObject*> {
public:
    List() {}
    void add(IOObject *obj)
    { list<IOObject*>::push_back(obj); }
};
```

במקרה זה, אין צורך לפרט את הפונקציות של list<IOObject*> בממשק של List, מכיון שהמחלקה האחרונה יורשת בצורה ציבורית מ-list<IOObject*>. כך אנו מנצלים את העבודה בתבניות ואת עקרונות תכנות מוכוון אובייקטים. כשמשתמשים באובייקט מסוג List אפשר לקרוא בעזרת הפונקציה io_read_file את האובייקטים מקובץ אל הרשימה, ואפשר גם להשתמש בכל הפונקציונליות של הרשימה list<IOObject*>.

ירושה מרובה יעילה כאשר אין פונקציות בעלות אותו שם לשתי המחלקות הבסיסיות. אם לשתי המחלקות הבסיסיות יש פונקציות הזהות בשמן, מוטל על המחלקה היורשת להגדיר את הפונקציה המתאימה לכל זוג פונקציות כאלו. במיוחד נכון הדבר עבור פונקציות וירטואליות.

## 13.1.2 המרת ממשק אובייקט לקבלת פונקציונליות אחרת

נניח, שיש לנו רשימה וכעת אנו רוצים לממש מחסנית. **מחסנית** (stack) היא מבנה נתונים המכיל אובייקטים אחרים, שבו אובייקט שנמצא בראש המחסנית, הוא זה שהוכנס אליה אחרון. האובייקט שנמצא בראש המחסנית הוא גם הראשון שיוצא ממנה.



### 13.1.2.1 שימוש ברשימה

ניתן לכתוב מחסנית ללא שימוש בכל מחלקה אחרת. דבר זה נוגד את רוח תכנות מוכוון האובייקטים. הפתרון המתבקש למצב זה הוא שימוש במחלקה אחרת, תוך כדי צמצום הקוד שעלינו לכתוב עד למינימום האפשרי. ניתן להשתמש במחלקה רשימה כדי לממש את המחסנית, באופן הבא:

```
template <class T>
class Stack {
    list<t> list;
public:
    Stack() {}
    void push(const T &v)
    { list.push_back(v); }
    T pop()
    { list.pop_back(); }
    ...
};
```

אם כן, אנו מממשים את המחלקה מחסנית בעזרת המחלקה רשימה. הפונקציונליות הנדרשת ממומשת כפונקציות של שורה אחרת, הקוראות לפונקציות המתאימות של המחלקה list. הרשימה נמצאת בחלק הפרטי של המחסנית, ולכן אין למתכנת שמשתמש במחלקה זו כל אפשרות להשתמש בפונקציות אחרות מהפונקציות המוכרזות בממשק המחסנית. במקרה זה, שימוש בפונקציה כמו push_front יפר את תנאי המחסנית. לכן, חשוב להגדיר את האובייקט בו משתמשים, בחלק הפרטי של המתאם.

### 13.1.2.2 גמישות במימוש המחסנית

בפתרון שהצגנו יש עדיין בעיה, כי המחסנית ממומשת על ידי הרשימה בלבד. פתרון יותר טוב הוא לאפשר למשתמש לקבוע את המחלקה שמשמשת את המחסנית. בסעיף זה נכיר שתי דרכים להשגת גמישות כזו, שימוש בירושה ושימוש בתבניות.

#### 13.1.2.2.1 שימוש בירושה

כדי להשיג את הגמישות המבוקשת, ניתן להשתמש בירושה ובפונקציות וירטואליות. אפשר להפוך את אובייקט המחסנית למחלקה בסיסית אבסטרקטית, שיהיה לה הממשק הבא:

```
template <class T>
class StackImpl {
public:
    StackImpl() {}
    virtual void push(const T &v) = 0;
```

```
    virtual T &pop() = 0;
    ...
};

template <class T>
class Stack {
    StackImpl<T> *impl;

public:
    Stack(StackImpl<T> *impl_ptr)
    { impl = impl_ptr; }
    void push(const T &v)
    { impl->push(v); }
    T pop()
    { return impl->pop(); }
    ...
};
```

כדי לחסוך מקום בדוגמה, התעלמנו מבדיקת שגיאות (למשל אם המצביע impl למחלקה הבסיסית אינו nil). כשמשתמשים במחסנית, יש לתת לה אובייקט שיורש מהיישום הבסיסי של מחסנית, כלומר, אובייקט היורש מהמחלקה הבסיסית StackImpl.

```
template <class T>
class Listimpl : public StackImpl<T>
{
    list<T> lst;
public:
    Listimpl() {}
    void push(const T &v)
    { lst.push_back(v); }
    T pop()
    { return lst.pop_back(); }
};

Listimpl<Object> impl;
Stack<Object> stck(&impl);
```

לפני שימוש במחסנית, יש להגדיר אובייקט היורש מהמחלקה הבסיסית המופשטת StackImpl, ולהעביר למחסנית מצביע לאובייקט זה. המחסנית משתמשת באובייקט כדי לממש את הפונקציונליות של המחסנית. סידור זה מעט מגושם מבחינת המתכנת המשתמש במחלקה Stack. נוסף לכך, הפונקציות מקבלות את ערכיהן בזמן ריצה, ולכן יש שימוש במנגנון הפולימורפי לקריאה לפונקציות. למרות שמנגנון זה יעיל כמעט כמו קריאה רגילה לפונקציה, עדיף לפתור זאת בצורה אחרת, אם ניתן.

### 13.1.2.2.2 שימוש בתבניות

ניתן להשתמש בתבניות כדי לממש מתאמים. בפתרון הקודם היתה לנו מחלקה בסיסית שקבעה את ממשק המחלקות שיכולות לשמש כמימוש המחסנית. כאן, נשתמש במנגנון התבניות כדי לקבוע את ממשק האובייקט שמשמש למימוש המחסנית. אם האובייקט אינו ממלא ממשק זה, נקבל שגיאת הידור.

בהתאם לזה נגדיר כעת מחלקת תבנית, והאובייקט שמשמש למימוש המחלקה יהיה הפרמטר שלה.

```
template <class Container>
class Stack {
    typedef Container::value_type value_type;
    Container impl;
public:
    Stack() {}
    void push(const value_type &v)
    { impl.push_back(v); }
    value_type pop()
    { return impl.pop_back(); }
    ...
};
```

המחלקה מחסנית היא מחלקת תבנית, שפרמטר התבנית שלה הוא המכולה שישמש למימוש המחלקה. סוג האובייקטים המוכנס למחסנית נקבע על ידי סוג האובייקטים שיכולה המכולה (impl) להכיל. לכן, נשתמש במשפט typedef כדי להגדיר את סוג האובייקטים המוכנסים למחסנית, כזהה לסוג האובייקטים של המכולה.

כעת, השימוש במחסנית אלגנטי יותר ונעשה בצורה הבאה:

```
Stack< list<Object>  > s1;
Stack< vector<Object> > s2;

Object obj;
s1.push(obj);
s2.push(obj);
```

מצב זה נוח מאוד למתכנת המשתמש במחסנית. הוא אינו צריך להגדיר אובייקט למימוש המחסנית, אלא רק להגדיר אותה. השימוש בתבניות כאן דומה מאוד לשימוש בירושה ובפונקציות וירטואליות כדי לקבוע ממשק של מחלקה בסיסית. בשני המקרים, המחלקה שאנו מפתחים משתמשת בממשק כדי לממש פונקציונליות. כשמשנים את האובייקט המשמש למימוש, אנו משנים, באופן אוטומטי, את מבנה הנתונים ואת האלגוריתמים (הפונקציונליות של האובייקט).

## 13.1.3 המחלקה stack ב-STL

לאחר הדיון הקודם אפשר לראות את המחלקה **מחסנית** (stack) בספרייה הסטנדרטית, ולהיווכח שהיא מבוססת על העקרונות שתיארנו בסעיף הקודם. המחלקה מחסנית היא מחלקת תבנית שפרמטר התבנית קובע את מימוש המחלקה.

הצהרה של מחלקה זו נראית כך :

```
template <class Container>
class stack {
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
public:
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};
```

הפונקציה empty מחזירה ערך אמת אם המחסנית ריקה, ואחרת - ערך שקרי. הפונקציה size מחזירה את מספר האובייקטים במחסנית. הפונקציה top מאפשרת לבדוק את ראש המחסנית, ללא ביטול האובייקט הזה. את שאר הפונקציות הסברנו בסעיף הקודם.

כדי לממש מחסנית אפשר להשתמש במחלקות אלו : vector, list, או deque. לכל אחת מהמחלקות יש יתרונות וחסרונות. נזכור שווקטור הוא המתאים ביותר למימוש מחסנית. כדי להבין את הטענה האחרונה, נסביר את החסרונות של כל אחד ממבני הנתונים למקרה זה. למקרים אחרים כמו תור, יכולה רשימה להיות מבנה הנתונים המועדף.

לכל צומת ברשימה מוחזקים שני מצביעים, לפנים ולאחור. כשמכניסים אובייקט לרשימה, יש להקצות צומת המכיל את האובייקט ואת המצביעים האלה. בנוסף, יש לתחזק את המצביעים כשמכניסים אובייקט לרשימה, או כשמבטלים מהרשימה. לכן, קיים בזבוז של מקום וזמן במימוש זה. היתרון של הרשימה בא לידי לביטוי כשצריך להכניס אובייקטים **באמצע** הרשימה ולא בקצוות. מבנה הנתונים **רשימה** מועדף למקרים בהם יש הרבה הכנסות אובייקטים באמצע הרשימה.

תור כפול (deque) דומה לווקטור, אך יש לו פונקציות המאפשרות הכנסת אובייקטים משני צידיו. יש לטפל בשני המקרים (הכנסה בתחילת התור או בסופו), ולכן יש צורך בעבודת ניהול נוספת, לה איננו זקוקים במקרה המחסנית. במחסנית מכניסים

אובייקטים רק לסופה ומבטלים אובייקטים רק מסופה. בנוסף, לתור כפול יש מפת בלוקים (ראה פרק 11), דבר שאינו קיים במחלקה וקטור. תור כפול מתחזק את מפת הבלוקים, יש על כך תשלום (קטן אמנם) בזמן, ולכן מבנה הנתונים האידיאלי הוא וקטור.

המחלקה וקטור יעילה מאוד בהכנסה והוצאה של אובייקטים לראש הווקטור. למחלקה זו אין מפה, ולכן אינה זקוקה לזיכרון נוסף כלשהו. מבנה הנתונים המועדף למחסנית הוא וקטור.

## 13.2 דוגמה - ניהול משימות (tasks)

בסעיף זה נראה דוגמה המשתמשת במחסנית ומשלבת ירושה ופונקציות וירטואליות. הדוגמה תכלול אוסף פעילויות אותן יש לבצע. כשניתנת הוראה לבצע את הפעילויות האלו הן מתבצעות, והאוסף שמכיל את הפעילויות מנוקה (הפעילויות בו נמחקות). הדוגמה עצמה מופיעה בקבצים: **adapt.cpp, task.cpp, task.h**.

### 13.2.1 היררכיית הפעילויות

כדי לאפשר גמישות למערכת התוכנה המטפלת בפעילויות אלו, נגדיר מחלקה בסיסית שמייצגת פעילות בסיסית, Task. קוד המחלקות המייצגות פעילויות נמצא בקבצים **task.h, task.cpp**.

#### 13.2.1.1 המחלקה Task

המחלקה Task היא מחלקה מופשטת שמשמשת להגדרת ממשק שינוצל על ידי שאר חלקי התוכנה. בהגדרת מחלקת ממשק כזו אין צורך ששאר חלקי התוכנה ידעו על סוגי פעילויות שונים, כלומר, על אובייקטים היורשים מ-Task. הדבר מאפשר למתכנת אחר להוסיף סוגי פעילויות אחרים מבלי לשנות את החלק העיקרי של התוכנה.

המחלקה Task מגדירה פונקציה doit המשמשת לביצוע פעילות ה-Task הנוכחי. המחלקה מגדירה פונקציות לכתיבה, או קריאה, של פעילות מאובייקט פלט, או קלט מסוג ostream או istream, בהתאמה.

פונקציה נוספת המגדירה מחלקה זו היא הפונקציה create, המכונה לעיתים **בנאי וירטואלי** (virtual constructor). כזכור, השפה אינה תומכת בהגדרת בנאים וירטואליים, ולכן, כדי ליצור אובייקט בצורה וירטואלית עלינו להשתמש בפונקציה וירטואלית, היוצרת אובייקט שזהה בסוגו לאובייקט קיים. כשנשתמש בפונקציה כזו לא נצטרך לדעת את סוג האובייקט, אלא רק להשתמש בפונקציה כדי ליצור אובייקטים.

```
class Task {
public:
    Task() {}
    virtual int doit() = 0;
    virtual void write(ostream &out) = 0;
    virtual void read(istream &in) = 0;
    virtual Task *create() = 0;
};
```

### 13.2.1.2 המחלקה CopyFileTask

מחלקה היורשת מהמחלקה הבסיסית ומייצגת פעילות כלשהי היא המחלקה CopyFileTask, אשר מעתיקה קובץ מקור לקובץ יעד כלשהו. המחלקה מכילה שני אובייקטים מסוג String המכילים את שמות הקבצים שעליהם היא פועלת. בנאי המחלקה מקבל את שמות הקבצים אלה, ומאתחל את האובייקטים src ו-dst לשמותיהם.

```
// Copy file task
class CopyFileTask : public Task {
    String src;     // source file name
    String dst;     // destination file name
public:
    CopyFileTask(const char *d, const char *s);
    int doit();
    void write(ostream &out);
    void read(istream &in);
    Task *create() { return new CopyFileTask("", ""); }
};
```

הפונקציה doit מגדירה שני אובייקטי קלט/פלט, in ו-out, המייצגים את הקבצים המבוקשים. היא מעתיקה תו אחרי תו מקובץ המקור לקובץ היעד. פעולה זו יקרה (כי כל תו נקרא ונכתב) כאשר היא מופעלת על קבצים ארוכים במיוחד. אני משאיר לקורא את חיפוש הדרך לשיפור פעולה זו.

```
// do the actual copy of src onto dst
int CopyFileTask::doit()
{
    ifstream in(src.get_cstr());
    ofstream out(dst.get_cstr());
    char c;
    if (in && out)
        while (in.get(c))
            out.put(c);
    return 1;
}
```

הפונקציה write מאפשרת לכתוב את האובייקט לקובץ, כך שאפשר יהיה לשחזר ממנו את אוסף הפעילויות במועד מאוחר יותר. הפונקציה כותבת את שמות הקבצים המעורבים בפעולת ההעתקה.

```
// writes the task into the given stream
void CopyFileTask::write(ostream &out)
{
   out << src.get_cstr() << space << dst.get_cstr()
       << new_line;
}
```

הפונקציה read קוראת את שמות שני הקבצים מתוך אובייקט הפלט שניתן לה. תחילה היא קוראת את המחרוזות לחוצץ בגודל מתאים, ולאחר מכן היא מעתיקה אותו לאובייקט היעד. גם פעולה זו ניתנת לשיפור, אשאיר זאת לקורא כתרגיל.

```
// reads the task from input
void CopyFileTask::read(istream &in)
{
    char buf[512];
    in >> buf;
    src = buf;
    in >> buf;
    dst = buf;
}
```

## 13.2.1.3 המחלקה WCTask

המחלקה WCTask מאפשרת לספור את מספר התווים, מספר המילים ומספר השורות בקובץ נתון. מחלקה זו מבצעת פעולה דומה לפקודה wc ב-UNIX. המחלקה שומרת באובייקט מסוג String את שם הקובץ שעליו יש לפעול.

```
// word counting task
class WCTask : public Task {
    String fname;   // file name
public:
    WCTask(const char *fn);
    int doit();
    void write(ostream &out);
    void read(istream &in);
    Task *create() { return new WCTask(""); }
};
```

לא נראה את קוד הפונקציות של המחלקה. הקורא מוזמן לבחון את הקובץ **task.cpp** שבדיסקט, כדי לקבל פרטים מלאים על הפונקציות שלה. הפונקציה doit סופרת את התווים, המילים והשורות בקובץ המבוקש. הפונקציות read ו-write כותבות וקוראות אובייקטים של מחלקה זו. הפונקציה create יוצרת אובייקט חדש מסוג WCTask.

## 13.2.2 מנגנון יצירת האובייקטים

כשכותבים אובייקטים לקובץ משתמשים במצביע לאובייקט נוכחי ובפונקציה וירטואלית כדי לכתוב אותו. מצב זה אינו קיים כשקוראים אובייקט מקובץ. כשקוראים את הקובץ, נתון רק שם המחלקה של האובייקט. לפי שם המחלקה יש ליצור אובייקט מתאים ולקרוא אותו בעזרת הפונקציה read של האובייקט החדש.

אם תיצור התוכנה אובייקט לפי שם אובייקט נתון, היא תכיר את האובייקטים ולכן לא תהיה גמישה ואי אפשר יהיה להרחיב אותה. לכן, עלינו ליצור מנגנון שיאפשר הוספה קלה של סוגי אובייקטים חדשים.

המנגנון ליצירת אובייקטים ממומש על ידי המחלקות MetaClass ו- class_vector.

### 13.2.2.1 המחלקה MetaClass

מחלקה זו מאפשרת לקבל מידע אודות האובייקט וליצור אובייקטים חדשים מסוג האובייקט שאליו היא מתייחסת. כזכור, המחלקה **type_info** מחזירה את שם המחלקה של אובייקט נתון. היא מוסיפה את שירות ייצור אובייקטים של אובייקטים ממחלקה נתונה בזמן ריצה.

מחלקה זו שומרת מצביע לאובייקט מסוג Task שבעזרתו היא תיצור אובייקטים חדשים בעת הצורך. בנוסף, נשמר שם המחלקה של האובייקט בשדה clname. שם המחלקה של האובייקט מתקבל על ידי הפעלה של האופרטור typeid המחזיר ייחוס לאובייקט מסוג type_info. קריאה לפונקציה name של אובייקט זה, מחזירה את שם המחלקה של האובייקט Task.

```
class MetaClass {
    Task *task;
    String clname;
public:
    MetaClass(Task *t) : clname(typeid(*t).name())
    { task = t; }
    MetaClass()
    { task = 0; }
    Task *create() { return (task ? task->create() : 0); }
    const char *name() const
    { return clname.get_cstr(); }
};
```

שם המחלקה של האובייקט משמש אותנו בקריאה ובכתיבה. בכתיבה, כדי להוסיף לכל אובייקט את שם המחלקה, לפני תכולת האובייקט. בעת קריאת אובייקטים משתמשים בשם המחלקה כדי ליצור את האובייקט המתאים בקריאה.

### 13.2.2.2 המחלקה class_vector

מחלקה זו היא מכולה של אובייקטים מסוג MetaClass. האובייקטים שנמצאים בווקטור זה הם אלה שהתוכנית מכירה ויכולה לכתוב, או לקרוא מקבצים. מחלקה זו מכילה וקטור של STL המשמש להחזקת האובייקטים MetaClass.

```
class class_vector {
    vector<MetaClass> vec;
    void init_classes();
public:
    class_vector() { init_classes(); }
    Task *create(const char *nm);
    void add_class(Task *tp)
    {  vec.push_back(MetaClass(tp)); }
};
```

הפונקציה init_class מוגדרת כך לשם הנוחות. היא יוצרת שני אובייקטים מסוג CopyFileTask ו-WCTask, ומכניסה אובייקטים מסוג MetaClass הקשורים אליהם. היינו יכולים לוותר על פונקציה זו, ולאפשר למשתמש בתוכנה להוסיף את המחלקות שהתוכנית משתמשת בהן בעזרת הפונקציה add_class.

```
void class_vector::init_classes()
{
    const char *const null = "";
    Task *tp = new CopyFileTask(null, null);
    vec.push_back(MetaClass(tp));
    tp = new WCTask(null);
    vec.push_back(MetaClass(tp));
}
```

הפונקציה create מאפשרת ליצור אובייקט חדש לפי שם המחלקה שלו. הפונקציה מקבלת את שם המחלקה של האובייקט החדש המבוקש וסורקת בלולאה את הווקטור ויוצרת את האובייקט הראשון בעל שם מחלקה המתאים לשם הנתון לה. אם לא קיים אובייקט כזה מוחזר אפס, כלומר, מצביע ריק.

```
// creates an object of a give class
Task *class_vector::create(const char *name)
{
    vector<MetaClass>::iterator i = vec.begin();
    while (i != vec.end()) {
       if (strcmp((*i).name(), name) == 0)
          return (*i).create();
       ++i;
    }
    return 0;
}
```

## 13.2.3 לב המערכת - אוסף המשימות

לב המערכת והמנהל שלה הוא המחלקה TaskColl, שמכילה את **המשימות** (tasks) שיש להפעיל. לעיתים מכנים את המשימות בשם **תהליכים**. מחלקה זו מכילה אוסף משימות מסוג stack. בנוסף, מכילה המחלקה וקטור של MetaClass, כלומר class_vector.

המחלקה TaskColl יורשת מ-Task ולכן ניתן להגדיר אוסף פעילויות לביצוע, ולהכניס אותן למחלקה TaskColl אחרת! דבר זה מאפשר להגדיר, בצורה רקורסיבית, פעילויות שמכילות פעילויות אחרות. צורה זו של ירושה היא **תבנית עיצוב** (design pattern). שיטת עבודה זו טובה כשצריך להגדיר מחלקות המכילות מחלקות בסיסיות, ויחד עם זאת יורשות ממחלקה בסיסית.

דוגמה לתבנית עיצוב יכולה להיות תפריט, שבו יכולות להיות שורות שהן תפריטים אחרים. לכן, ניתן להגדיר שתי מחלקות: MenuItem ו-Menu. המחלקה האחרונה יורשת מהמחלקה MenuItem ומכילה מספר רב של מחלקות מסוג MenuItem.

דוגמה אחרת לתבנית עיצוב היא קובץ וספריית קבצים. ספריית קבצים היא קובץ המכיל קבצים אחרים. אני משאיר לקורא המעוניין את פתרון בעיית הקבצים והספריות.

```
class TaskColl : public Task {
    stack<vector<Task*> > tasks;
    class_vector cvec;
public:
    TaskColl() {}
    int doit();
    void write(ostream &out);
    void read(istream &in);
    Task *create() { return new TaskColl(); }
    void add_task(Task *p)
    { tasks.push(p); }
    void write_file(const char *nm);
    int read_file(const char *nm);
};
```

הפונקציה doit מבצעת את כל הפעילויות שבאוסף הפעילויות. היא פועלת בלולאה על מחסנית הפעילויות עד שהמחסנית מתרוקנת. בכל שלב של הלולאה נשלף אובייקט אחד מהמחסנית ונקראת הפונקציה doit של האובייקט, ולאחר מכן משוחרר האובייקט.

הפונקציה top מחזירה ייחוס לאובייקט שבראש המחסנית, ולכן יש לבדוק שהמחסנית אינה ריקה **לפני** שמשתמשים בפונקציה. הפעלת הפונקציה על המחסנית ריקה אינה מוגדרת ובמקרה הטוב התוכנית תופסק. במקרה הרע, תמשיך התוכנית לפעול, אך בצורה מוזרה ובלתי מוגדרת.

```
int TaskColl::doit()
{
    int n = 0;
    while (!tasks.empty()) {
      Task *tp = tasks.top();
      tp->doit();
      ++n;
      delete tp;
      tasks.pop();
    }
    return n;
}
```

הפונקציה write מאפשרת לכתוב את הפעילויות המבוקשות לתוך אובייקט קלט. כמו הפונקציה הקודמת, כך גם זו פועלת בלולאה על כל פעילויות המחסנית. בכל שלב של הלולאה נשלף האובייקט בראש המחסנית ונכתב לאובייקט הקלט הנתון לפונקציה. כשנכתב אובייקט לקובץ, נכתב תחילה שם המחלקה של האובייקט, ולאחר מכן האובייקט עצמו. שם המחלקה של האובייקט משמש לאחר מכן בעת קריאת הקובץ, ליצירת אובייקט מתאים.

```
// writes the task collection into a given stream.
void TaskColl::write(ostream &out)
{
    while (!tasks.empty()) {
         Task *tp = tasks.top();
         out << typeid(*tp).name() << space;
         tp->write(out);
         delete tp;
         tasks.pop();
    }
}
```

הפונקציה write_file מכינה את אובייקט הפלט ומחברת אותו לקובץ המבוקש. אם פתיחת הקובץ הצליחה, נקראת הפונקציה הקודמת עם אובייקט הפלט.

```
void TaskColl::write_file(const char *nm)
{
    ofstream out(nm);
    if (out)
      write(out);
}
```

הפונקציה read פועלת בלולאה כשבכל שלב היא קוראת את שם המחלקה של האובייקט הנוכחי. הפונקציה בודקת אם יש מחלקה עם שם המתאים לשם שנקרא מהקובץ. אם אכן מוחזר מצביע לאובייקט חדש שנוצר כתוצאה מקריאה לפונקציה create של class_vector, נקרא האובייקט מהקלט ומתוסף לאוסף הפעילויות לביצוע.

```
void TaskColl::read(istream &in)
{
    while (in) {
         char name[128];
         in >> name;
         if (!in)
              break;
         Task *tp = cvec.create(name);
         if (tp) {
              tp->read(in);
              add_task(tp);
         }
    }
}
```

הפונקציה read_file מגדירה אובייקט קלט הקשור לקובץ הנתון. אם האובייקט תקין, כלומר, הקובץ המבוקש קיים והאובייקט הצליח לפתוח אותו, אזי הפונקציה משתמשת בפונקציה read כדי לקרוא את הקובץ.

```
int TaskColl::read_file(const char *nm)
{
    ifstream in(nm);
    int res = 0;
    if (in) {
         res = 1;
         read(in);
    }
    return res;
}
```

## 13.2.4 פונקציות הניהול

פונקציות הניהול כוללות את הפונקציה הראשית של התוכנית ופונקציות עזר, המאפשרות לקרוא פרטים מהקלט הסטנדרטי של התוכנית.

הפקודות שהתוכנית מאפשרת הן: הוספת פעילות, ביצוע אוסף פעילויות, כתיבה או קריאה של אוסף פעילויות מתוך קובץ. פקודות אלו מפורטות בפקודה enum הבאה:

```
enum commands { add_copy_cmd, add_wc_cmd, doit_cmd,
                          read_cmd, write_cmd, exit_cmd };
```

הפונקציה display_menu מציגה תפריט למשתמש הכולל את הפקודות שמבצעת התוכנית. פקודות אלו מפורטות בפקודה enum הקודמת. הפונקציה מחזירה את האפשרות שנבחרה בתפריט.

```
int display_menu()
{
   int res;
   cout << add_copy_cmd << ") add a copy file task."
        << new_line;
   cout << add_wc_cmd << ") add a word counting task."
        << new_line;
   cout << doit_cmd << ") perform tasks." << new_line;
   cout << read_cmd << ") read tasks from a file."
        << new_line;
   cout << write_cmd << ") write tasks to a file."
        << new_line;
   cout << exit_cmd << ") exit." << new_line;
   cout << "> ";
   cin  >> res;
   cout << new_line;
   return res;
}
```

הפונקציה add_copy_task מוסיפה פעילות מסוג CopyFileTask לאוסף הפעילויות הנתון לה. היא קולטת מהמשתמש את שמות הקבצים המבוקשים, יוצרת פעילות חדשה ומכניסה אותה לאוסף הפעילויות.

```
void add_copy_task(TaskColl &coll)
{
   char name1[256], name2[256];
   cout << "\nenter source and destination: ";
   cin >> name1 >> name2;
   coll.add_task(new CopyFileTask(name2, name1));
}
```

הפונקציה get_name מאפשרת לקלוט שם מהמשתמש, אל המערך name. מערך זה מוגדר כסטטי, כדי שימשיך להתקיים לאחר חזרת הפונקציה. אחרת היה המערך מוגדר במחסנית התוכנית, והיה נעלם כשהפונקציה היתה חוזרת. הגדרת מערך, או אובייקט אוטומטי כלשהו והחזרה שלו כתוצאה של פונקציה, היא באג שכיח של מתכנתים רבים. תוכנית המבצעת פעולות כאלו מגלה התנהגות מוזרה בדרך כלל, כי באגים אלה אינם קלים לגילוי וזיהוי.

```
char *get_name(const char *prompt)
{
   static char name[512];
   cout << new_line << prompt;
   cin >> name;
   return name;
}
```

הפונקציה add_wc_task מוסיפה פעילות מסוג WCTask לאוסף הפעילויות הנתון. שם הקובץ הקשור לפעילות זו מתקבל על ידי קריאה לפונקציה get_name.

```
void add_wc_task(TaskColl &coll)
{
        char *name = get_name("enter file name:");
        coll.add_task(new WCTask(name));
}
```

הפונקציה הראשית של התוכנית פועלת בלולאה, שבכל שלב שלה מוצג למשתמש תפריט של פעולות אפשריות בעזרת display_menu. הלולאה מסתיימת כשהפונקציה display_menu מחזירה את הערך exit_cmd. בשאר המקרים הפונקציה משתמשת בפונקציות אחרות שהוגדרו בקובץ, כדי לבצע את האפשרות שנבחרה מהתפריט.

```
int main()
{
    TaskColl tasks;
    char *name;
    int cmd;
    do {
        cmd = display_menu();
        switch (cmd) {
        case add_wc_cmd:
               add_wc_task(tasks);
               break;
        case add_copy_cmd:
               add_copy_task(tasks);
               break;
        case doit_cmd:
               tasks.doit();
               break;
        case write_cmd:
               name = get_name("enter output file name:");
               tasks.write_file(name);
               break;
        case read_cmd:
               name = get_name("enter input file name:");
               tasks.read_file(name);
               break;
        default:
               break;
        }

    } while (cmd != exit_cmd);
    return 0;
}
```

לסיכום, התוכנית מאפשרת לבצע דברים רבים: להוסיף פעילויות, לכתוב אותן לקובץ, או לקרוא אותן מקובץ. התוכנה המטפלת בפעילויות היא גנרית, לכן אפשר להרחיב את המערכת בצורה קלה עם סוגי פעילויות אחרות (ראה תרגילים). כדי להרחיב את המערכת בפעילות נוספת יש לגזור את הפעילות החדשה מהפעילות הבסיסית ולהוסיף את האחרונה למחלקה class_vector.

## 13.3 סיכום

בפרק זה למדנו כיצד לשלב **תכנות מוכוון אובייקטים** (object oriented programming) עם **תכנות גנרי** (generic programming) המבוסס על **תבניות** (templates). שני כלים אלה הם בעלי עוצמה רבה וכשמשלבים את שניהם מתקבלת האפשרות לכתוב מערכות גנריות שאינן תלויות בסוג האובייקטים עליהן הן פועלות.

מונח אחר שלמדנו בפרק זה הוא **מתאם** (adaptor). מתאם מאפשר לשנות את הממשק של אובייקט מסוים לממשק אחר. כך אפשר להשתמש בפונקציונליות של האובייקט המתואם במערכת המצפה לממשק אחר.

שימוש אחר של מתאמים הוא יצירת ממשק חדש מפונקציונליות נתונה של אובייקט אחר. למשל, יצירת מחסנית יכולה להיות ממומשת על ידי רשימה, וקטור או תור כפול. אם נאפשר לממשה על ידי אחד מאלה, לא נצטרך לכתוב קוד נוסף של מחסנית.

אפשר להגדיר את היישום של המחסנית על ידי מחלקה בסיסית, ולהגדיר מספר מחלקות שונות לכל יישום שהוא. כשמגדירים אובייקט מחסנית יש למסור לו את היישום שמתכוונים אליו.

אפשרות אחרת, עדיפה במקרה זה, היא לאפשר למתכנת המשתמש במחלקה מחסנית, לקבוע את המימוש על ידי פרמטר של תבנית. הדבר מבטל את המנגנון הפולימורפי, ואת הצורך בירושה.

ראינו בפרק זה שילוב בין תכנות גנרי לתכנות מוכוון אובייקטים. יצרנו מערכת פולימורפית שהשתמשה בפונקציות גנריות המבצעות אוסף פעילויות. המערכת גמישה וניתנת להרחבה על ידי הגדרת פעילויות חדשות, היורשות מהפעילות הבסיסית.

## 13.4 שאלות

1. ממש תור בעזרת מכולה אחרת כלשהי של STL. האם אפשר לממש את המחסנית בעזרת וקטור?
2. למתכנת המשתמש בתור מהשאלה הקודמת, באיזה מבנה נתונים היית ממליץ להשתמש למימוש התור?
3. שפר את הפונקציה doit של המחלקה CopyFileTask (החיסרון של פונקציה זו הוא הקריאה של כל תו בנפרד, שפירושו קריאות רבות לפונקציה).

4. הפונקציה read של CopyFileTask גורמת לשתי העתקות. אחת לחוצץ מקומי ואחת לזיכרון פנימי של המחלקה String. שפר פעולת קריאה מהמחלקה String באופן כזה שלא תהיינה שתי העתקות.

5. ממש את התוכנית המטפלת ב-Tasks בעזרת התור מהשאלה הראשונה.

6. הוסף פעילות של ספירת מספר הקבצים בספרייה נתונה למערכת של פרק זה. איזו פונקציה עליך להוסיף למחלקה TaskColl, כדי לאפשר תוספת קלה של אובייקטים, מבלי לשנות את הפונקציה init_classes.

נספח

# דיסקט התוכניות המצורף לספר

## 1 מה בדיסקט

בדיסקט המצורף תמצא:

- תוכניות שהדגמנו בספר
- ספריית תבניות סטנדרטית של ++C בשם STL

## 2 התקנה

הכנס את הדיסקט לכונן A או B.

הפעל את קובץ INSTALL מהדיסקט המצורף

a:\>install או b:\>install

אם אתה נמצא ב- Windows 95 או ב-Windows 3.11 - לחץ לחיצה כפולה על שם הקובץ INSTALL (בסייר Windows או במנהל הקבצים, בהתאמה).

ההתקנה פותחת את הספרייה CPPFC בדיסק הקשיח ומעתיקה אליה את התוכניות ואת הספרייה STL שבדיסקט.

## 3 הרצה - כללי

אפשר להריץ את התוכניות **במהדר בורלנד מגירסה 4 ומעלה**, או **במהדר מיקרוסופט ++Visual C גירסה 4.2 ומעלה**.

את התוכניות שבספר, ואשר חלקן נמצאות בדיסקט, ניתן להדר ולהריץ תחת **חלונות**, או בסביבת העבודה של **DOS**.

שפת ++C אינה תלויה בסביבת העבודה, ולכן התוכניות שבספר זה יוכלו לפעול בכל סביבה שבה יש מהדר ++C מתקדם מספיק. לתכנות והידור התוכניות השתמשנו בשני מהדרים: במהדר בורלנד גירסה 4.5 והשתמשנו גם במהדר של סאן (Sun) גירסה 4.2. בנספח זה נסביר כיצד אפשר להדר את התוכניות בשתי סביבות העבודה האלו בעזרת כלים המסופקים על ידי חברת בורלנד.

# 4 הידור והרצה תחת חלונות

עבור ל-Windows. כדי להיכנס לסביבת העבודה של בורלנד בחר בעזרת העכבר את הסמל המתאים במנהל היישומים או בשולחן העבודה.

## 4.1 הגדרת פרויקט

בסביבת הפיתוח של בורלנד בחר מתפריט Project את הפריט New Project. כתוצאה מבחירה זו מתקבל תת-חלון שבו יש להקליד את שם הפרויקט המבוקש. נניח שאנו רוצים להריץ תוכנית פשוטה, כמו זו:

```
#include <iostream.h>

int main()
{
	cout << "\nhellow world" << endl;
	return 1;
}
```

נניח ששם הפרויקט שלנו הוא hello. לפיכך, שם הפרויקט המלא (כולל שם הנתיב) יהיה למשל:

```
c:\cppfc\hello.ide
```

שם ה-Target מתעדכן באופן אוטומטי כאשר תקליד את שם הפרויקט, ובמקרה זה הקלד hello. בתת-החלון בחר את סוג המטרה (Target Type). מכיון שזו תוכנית פשוטה שאינה משתמשת בספריית האובייקטים לחלונות, בחר: EasyWin. את שאר האופציות ניתן להשאיר לפי ברירת המחדל. לאחר מכן בחר Apply בעזרת הלחצן השמאלי של העכבר. תת-החלון יעלם ויופיע חלון הפרויקט, שיכיל את השם hello.c.

ניתן לבטל או להוסיף קבצים לפרויקט על ידי בחירה של פריט בחלון הפרויקט ולחיצה על הלחצן הימני של העכבר. כתוצאה מלחיצה זו יופיע תפריט שקיימים בו מספר פריטים וביניהם פריטים שמאפשרים להוסיף או לבטל קובץ: Add Node, Delete Node. כדי לבטל את הקובץ hello.c בחר את הפריט Delete Node. כדי להוסיף את הקובץ hello.cpp לחץ על הלחצן הימני של העכבר בתוך חלון הפרויקט. לחיצה זו

תעלה תפריט שיכלול Add Node. בחירה של פריט זה מאפשרת להוסיף שם קובץ, למשל hello.cpp.

לאחר הוספת הקובץ יופיע פריט בשם hello.cpp בתת-חלון הפרויקט. שתי לחיצות מהירות על הלחצן השמאלי יפתחו חלון עם עורך שבו תוכל להקליד את התוכנית. לאחר עריכת התוכנית תוכל לפנות להידור התוכנית.

## 4.2 הידור התוכנית

לחץ על הלחצן השמאלי של העכבר כשהסמן נמצא על תפריט Project. כתוצאה יוצג תפריט המכיל את הפריט Build All. בחירה של הפריט האחרון תבנה את הפרויקט.

## 4.3 הרצת התוכנית

בחר בתפריט Debug והפריט Run. לחילופין, תוכל להריץ את התוכנית על ידי פניה לחלון Program Manager, בחירה של תפריט File ופריט ...Run מתוכו, תקבל תת-חלון שבו יש להקליד את שם התוכנית והארגומנטים שלה להרצה. לאחר מכן לחץ על לחצן OK לאישור הפעולה.

# 5 הידור והרצה בסביבת DOS

להרצה בסביבת DOS תמצא בדיסקט את הקובץ makefile, אשר מכיל פקודות לבניית יישום (make).

## 5.1 חוקים בסיסים של make

**שים לב!** החוקים המתוארים כאן הם עבור היישום make של בורלנד!

בקובץ make אפשר להגדיר מקרו בצורה הבאה:

```
macro-name = value
```

כלומר, שם המקרו מוגדר ומקבל ערוץ. במקרו כזה אפשר להשתמש בכל מקום בקובץ, בצורה הבאה:

```
$(macro-name)
```

יש להוסיף את התו $ בתחילה, ואת שם המקרו יש לכתוב בין סוגריים. בעזרת המקרו אפשר להגדיר את הסביבה בה עובדים, ולפיכך אפשר לשנות את ערך המקרו כשעוברים לסביבה אחרת, ללא שינויים רבים בקובץ make.

בקובצי make יש מטרות שהן משימות שהתוכנית make יודעת לבצע. מטרה היא שם כלשהו ואחריו מופיע התו ":". המטרה תלויה ברשימה המופיעה אחריה באותה שורה. בשורה שבאה לאחר מכן מוגדר מה לעשות במקרה והמטרה אינה מעודכנת. למשל בקובץ אשר מצורף לספר מופיעה המטרה הבאה:

```
list_t.exe: list.obj
      $(CC) -L$(LIBSDIR) $(CFLAGS) list_t.cpp list.obj
```

במקרה זה, המטרה list_t.exe תלויה בקובץ list.obj. אם הזמן האחרון שהקובץ list.obj השתנה מאוחר מאשר הזמן של הקובץ list.exe, יחשב הקובץ האחרון כלא מעודכן, ולכן תבוצע הפקודה בשורה לאחר מכן. בשורה השנייה משתמשים בערך של המקרו CC כדי להדר את הקבצים שיוצרים את התוכנית list_t.exe. בשורה זו יש שימוש בשני מקרו נוספים המכילים דגלים למהדר: LIBSDIR, CFLAGS. כמו בשורה זו מופיעים שמות הקבצים שיש להדר.

לאחר שראינו את החוקים הבסיסים, אנו יכולים להבין מספר מקרוס שנמצאים בקובץ make בדיסקט המצורף.

המקרו CCDIR=c:\bc4 מגדיר את המיקום של המהדר. אם המיקום של המהדר שברשותך אינו מתאים לנתיב הזה, עליך לרשום את המיקום הנכון שלו.

מקרו אחר שיהיה עליך לשנות הוא המקרו המגדיר את דגלי המהדר:

```
CFLAGS = -v -m$(MODEL) -Istl
```

מקרו זה מגדיר את המיקום של STL (הספרייה הסטנדרטית של C++ שנמצאת בדיסקט זה). במחשב שבו הרצנו את התוכניות, שם הספרייה הוא stl. הגדרת מיקום הספרייה מופיע בצמוד ל- "I-". עליך לספק את השם המלא (כולל שם הנתיב) של הספרייה, אם היא אינה נמצאת כתת-ספרייה של ספריית התוכניות. בדוגמה זו stl נמצאת כתת-ספרייה של ספריית התוכניות ולכן אפשר לכתוב את שמה בלבד.

**הערה**: בהתקנת הדיסקט יצרנו את הספרייה STL בתוך הספרייה CPPFC שבה נמצאות התוכניות.

## 5.2 הידור והרצה

כדי להדר הוספנו מטרה all בקובץ make. מטרה זו תלויה בכל התוכניות המצורפות. לפיכך, כדי להדר את כל התוכניות במערכת הפעלה DOS, צריך להקליד: make all. כך יתקבלו התוכניות בקבצים בעלי סיומת exe, כמו למשל, הקובץ list_t.exe (מפרק 5). את התוכנית ניתן להריץ על ידי הקלדת השם בלבד בשורת הפקודה, ובמקרה זה: list_t.

# אינדקס עברי

## ב



## ג

## ד



## ה

## ו



## ז



## ח



## ט



## י

## צ

## ק



## ר

## ש



## ת

# אינדקס לועזי

**T**



**U**



**V**

הוצאת הוד-עמי ספרי מחשבים
הראובני 6, ת.ד. 6108 הרצליה 46160
טלפון: 09-9564716 פקס: 09-9571582
info@hod-ami.co.il www.hod-ami.co.il

# קטלוג אוקטובר 2000

| | | עמ' | | מחיר* |
|---|---|---|---|---|
| **OFFICE 2000** | | | | |
| קוראים < יודעים **OFFICE 2000** | | 616 | CD | 149 |
| קוראים < יודעים **WORD 2000** | | 592 | CD | 129 |
| קוראים < יודעים **POWERPOINT 2000** | | 104 | | 39 |
| קוראים < יודעים **EXCEL 2000** | | 512 | CD | 139 |
| **WORD 2000** ישר ולעניין | | 348 | | 99 |
| **OFFICE 2000** ישר ולעניין | | 384 | | 99 |
| **WORD 2000** תכליס - צעד אחר צעד | | 240 | CD | 79 |
| **EXCEL 2000** תכליס - צעד אחר צעד | | 240 | CD | 79 |
| **OFFICE 2000** תכליס - צעד אחר צעד | | 560 | CD | 119 |
| המדריך השלם **ACCESS 2000 VBA** | | 640 | CD | 189 |
| **ACCESS 2000** סדנת לימוד | חדש | 384 | CD | 139 |
| **אינטרנט/גרפיקה באינטרנט** | | | | |
| **XML** למפתחי אתרים באינטרנט | **בקרוב** | | CD | 149 |
| **Director 8** למפתחי אתרים באינטרנט | **בקרוב** | | CD | 149 |
| **HTML 4** למפתחי אתרים באינטרנט | **מהד' 3** | 592 | CD | 159 |
| **Java 2** למפתחי אתרים באינטרנט | | 432 | CD | 99 |
| **JavaScript 1.2** למפתחי אתרים באינטרנט | | 192 | CD | 79 |
| **FLASH 4** למפתחי אתרים באינטרנט | **מהד' 2** | 368 | CD | 149 |
| **ASP 3** למפתחי אתרים באינטרנט | | 248 | CD | 119 |
| **ASP 3 ובניית אתרים ב- Visual InterDev** סדנת לימוד | חדש | 352 | CD | 129 |
| **MP3** מוסיקה באינטרנט עם Winamp | | 224 | CD | 99 |
| **Paint Shop Pro 6** גרפיקה באינטרנט | | 480 | CD | 129 |
| קוראים < יודעים אינטרנט עם **Internet Explorer 5.x** | **מהד' 2** | 224 | CD | 69 |
| לפטפט ברשת עם **mIRC** | | 144 | CD | **35** |
| **שפות תכנות** | | | | |
| ערכת כלים **Visual Basic 6** | | 424 | CD | 119 |
| סדנת לימוד **Visual Basic 6** | **מהד' 2** | 1088 | CD | 249 |
| Visual Basic 6 **תכנות משחקי מחשב** | **מהד' 2** | 264 | CD | 95 |
| **GUI** פיתוח ממשק משתמש בסביבת Windows | חדש | 544 | CD | 249 |
| **Win32API** ומבוא ל- **MFC** למתכנתי Visual C++ 6 | חדש | 656 | CD | 249 |
| סדנת לימוד **Visual C++ 6** | | 992 | CD | 249 |
| המדריך השלם **Visual C++ 6** | | 1096 | CD | 249 |
| סדנת לימוד **C++** | **בקרוב** | | CD | |
| C++ בקלות + מהדר Borland Turbo | **מהד' 2** | 436 | ☑ | 79 |
| **C++ בקלות** + מהדר Borland Turbo | **מהד' 3** | 480 | CD | 119 |
| **ללמוד C** + מהדר Borland Turbo | **מהד' 2** | 480 | CD | 119 |
| שפת **C - נושאים מתקדמים ומולטימדיה** | חדש | 432 | CD | 139 |

| | | עמ' | | מחיר* |
|---|---|---|---|---|
| המדריך השלם לשפת **C** + מהדר Borland Turbo | **מהד' 6** | 520 | ☑ | 135 |
| שפת **C** - תוכניות ופתרונות | **מהד' 2** | 384 | **CD** | 99 |
| שפת **C** מעבר לשיא - ספר לימוד | **בקרוב** | | **CD** | 119 |
| שפת **C** מעבר לשיא - ספר תרגילים | **בקרוב** | | **CD** | 99 |
| שפת אסמבלי למחשב האישי | **מהד' 2** | 352 | **CD** | 89 |
| פסקל מהצעד הראשון | | 256 | | 59 |
| **PC - חומרה, תוכנה וניהול** | | | | |
| הסדרה הידידותית **הכרת המחשב האישי** - מהד' 2 | | 128 | | 49 |
| המדריך השלם ל**טכנאי PC** - **חומרה ותוכנה** (דורון סיון) | | 496 | **CD** | 149 |
| המדריך השלם ל**טכנאי PC** - **רשתות תקשורת** (דורון סיון) | | 552 | **CD** | 149 |
| המדריך השלם ל**טכנאי PC**-**רשתות תקשורת** (דורון סיון) **- מהד' 2** | **חדש** | 712 | **CD** | 189 |
| המדריך הכפול ל**טכנאי PC** - **חומרה** + **רשתות (מהד' 1)** | | 1048 | **2CD** | 239 |
| **מבחני הסמכה** | | | | |
| הכנה למבחן הסמכה **Windows 2000 Server** | **חדש** | 1104 | **CD** | 359 |
| MCSE Readiness Review - **Windows NT 4.0** | | 456 | **CD** | 99 |
| הכנה למבחן הסמכה **Networking Essentials** | **מהד' 3** | 728 | **CD** | 359 |
| הכנה למבחן הסמכה **A+** לטכנאי PC | | 1016 | **CD** | 359 |
| **שונות** | | | | |
| **מילון** הוד-עמי **למונחי מחשב** בשיתוף מכון התקנים | **חדש** | 768 | **CD** | 189 |
| **פריצה**! לא במחשב שלי! | **בקרוב** | | | 99 |
| ניהול איכות תוכנה | | 312 | | **49** |
| סדנת לימוד **Project 98** | | 528 | **CD** | **99** |
| ספר התמיכה Help Desk | | 288 | | **49** |
| **מחוללי יישומים ובסיסי נתונים** (Access ראה בעמוד קודם) | | | | |
| מחסני נתונים | | 352 | | **89** |
| ארגון נתונים וקבצים | | 396 | | 84 |
| בסיסי נתונים טבלאיים ושפת SQL - עקרונות ועיצוב | | 672 | **CD** | 149 |
| בסיסי נתונים טבלאיים ושפת SQL | | 480 | ☑ | 89 |
| **תקשורת** | | | | |
| **תקשורת מחשבים** - פרוטוקולים וארכיטקטורות רשת | | 496 | **CD** | **89** |
| רשת נובל - **NetWare 5** - מדריך הפעלה ושירות | | 584 | **CD** | **119** |
| רשת נובל - NetWare 4.1 כרך א' | | 336 | | 119 |
| **Office 97** | | | | |
| קוראים > יודעים **Office 97** | | 672 | **CD** | 149 |
| **Excel 97** ישר ולעניין | | 260 | | 99 |
| ***הסדרה הידידותית*** **Excel 97** | | 478 | ☑ | 115 |
| ***הסדרה הידידותית*** מה חדש ב - **Word 97** | | 180 | | **39** |
| ***הסדרה הידידותית*** **PowerPoint 97** | | 112 | | 49 |
| **Word 97** תכליס (למתחילים) | | 200 | ☑ | 69 |
| **Excel 97** תכליס (למתחילים) | | 224 | ☑ | 69 |
| **Access 97** ישר ולעניין | | 276 | | 99 |
| סדנת לימוד **Access 97 VBA** | | 240 | **CD** | 79 |

| | עמ' | | מחיר* |
|---|---|---|---|
| **מערכות הפעלה** | | | |
| **קובץ אצווה** | 52 | | 20 |
| **Linux העידן החדש + תוכנת Linux RedHat - גירסה 6** | 352 | CD | 149 |
| **WINDOWS** | | | |
| **Millennium** Windows הסדרה הידידותית — חדש | 144 | | 49 |
| קוראים > יודעים **Windows 2000** | 512 | CD | 149 |
| קוראים > יודעים **Windows 98** — **מהד' 2** | 592 | CD | 125 |
| **Windows 98** למשתמשי גרסאות קודמות | 416 | CD | **59** |
| ניהול מערכת **Windows 98** - פיטר נורטון | 680 | CD | **99** |
| **Windows 98** ישר ולעניין | 312 | | 119 |
| **Windows 98** תכליס (למתחילים) | 240 | ☑ | 69 |
| **Windows 95** תכ'לס (למתחילים) | 200 | ☑ | 69 |
| *הסדרה הידידותית* **Windows 95** — **מהד' 2** | 528 | | 115 |
| **גרפיקה** | | | |
| **Paint Shop Pro 6** גרפיקה באינטרנט | 480 | CD | 129 |
| עיצוב וגרפיקה עם **Paint Shop Pro 5.x** | 384 | CD | 119 |
| עיצוב וגרפיקה עם **PhotoShop 5** | 448 | CD | 149 |
| **תוכנות Office** לחלונות 3.11 **ו- Office 7** | | | |
| *הסדרה הידידותית* **Word 7** for Windows 95 + **CD** מבחן אישי | 496 | CD | **59** |
| *הסדרה הידידותית* **Excel 7** for Windows 95 | 464 | | **49** |
| *הסדרה הידידותית* **Excel 7 VBA** for Windows 95 | 464 | | **49** |
| **Excel 7** מהצעד הראשון + תוכנת X-Tools v2 | 240 | ☑ | **59** |
| **Word 7** במשרד הממוחשב | 248 | ☑ | **39** |
| **Word 6** בבית-הספר | 144 | ☑ | 39 |
| **Word 6** למד להצליח - באישור משרד החינוך | 304 | ☑ | **49** |
| **ביוגיליון** - ביולוגיה בעזרת אקסל | 288 | ☑ | **49** |

*** המחירים בש"ח כולל מע"מ ומשלוח חינם**